CONTEXT-AWARE PROGRAMMING LANGUAGES

TOMAS PETRICEK

May 2016

Clare Hall, University of Cambridge

This dissertation is submitted for the degree of Doctor of Philosophy.

DECLARATION

This dissertation is my own work and includes nothing which is the outcome of work done in collaboration with others except where specifically indicated in the text. This dissertation does not exceed the regulation length of 60,000 words, including tables and footnotes.

ABSTRACT

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

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

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

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

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

CONTENTS

Part I

CONTEXT-AWARE PROGRAMMING

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

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

We start with a detailed discussion of the motivation for the thesis and an overview of our methodology (Chapter 1). Next, we discuss previous programming language research that leads to the work presented in this thesis (Chapter 2) and we examine a number of practical context-aware systems in detail (Chapter 3), identifying two kinds of context that we later capture by *flat* and *structural coeffects*.

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 exhaustive, but they are representative of the various kinds of motivations.

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 [45], data-parallelism and also asynchronous computing [105].

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" is an example of a more subtle challenge. Initiatives like open government data make more data available, but 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 [65] 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[1], 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[2].

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*.

1 The dominant approach is demonstrated, for example, by a review of SQL injection prevention techniques by Clarke [23]

2 This is difficult to explain in writing and so the reader is encouraged to watch a video showing type providers for the WorldBank and CSV data sources [82].

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.

THE THESIS. The key claim of this thesis is that the theory of *coeffects* that we develop provides a unified way of capturing how programs rely on the context, or execution environment, in which they execute. Perhaps most importantly, coeffects provide a single conceptual framework that can capture a wide range of previously disconnected notions of context-awareness.

Thanks to coeffects, programming languages for rich and diverse execution environments will be able to offer a simple, safe and unified programming model for interacting with the environment. In this thesis, we develop the theoretical foundations for such languages and we develop a prototype implementation that illustrates the benefits of coeffects, but also serves as a rich guide for future langauge implementers.

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 [65] let developers write code in a host language like C# and then cross-compile it to multiple targets (including SQL, OpenCL or JavaScript [61]). 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 statically – if a program attempts to access unavailable data, the access will be refused at run-time.

- RESOURCES & DATA AVAILABILITY. When creating a mobile application, the program may (or may not) be granted access to device capabili-

```
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
    | "referer" → req.Headers.[HttpHeader.Referer] ← value
#endif
    | other → req.Headers.[other] ← value
```

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

ties 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, depending on the role of the user.

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 – the 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 [41] 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[3]. The example uses conditional compilation to target multiple versions of the .NET framework. Such code is difficult to write – to see whether a change is correct, it had to be recompiled for all combinations

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

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[4] 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 can provide a better solution.

1.1.2 *Context awareness #2: System capabilities*

Another example related to the previous one is when libraries use metaprogramming techniques, such as LINQ [65, 22] or F# quotations [102], 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 broadly applicable technique for programming in heterogeneous environments. It lets developers target 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 an approach where code written using libraries for one execution environment is translated and run in another, more limited, execution environment.

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 11,800 results for the message above and even more results (44,100) for a LINQ error message *"Method X has no supported translation to SQL"* caused by a similar limitation.

4 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 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* [23]. An attacker could enter `"'; DROP TABLE Products --"` as the `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.

Again, this example demonstrates a more general property. Sometimes, it is desirable to track additional metadata about values that are in some ways special. Such metadata can determine how the values can be used. Here, the value stored in `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.

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* [44], but the technique is a special case of more general *provenance tracking* [21]. This can be useful when working with data in other contexts. For example, data journalists might want to propagate metadata about the quality and the information source – is the source trustworthy? Is the data up-to-date? Such metadata 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 dataflow programming examples used later in the thesis.

Consider a simple programming language with arrays where the 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).

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 [77]. 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 metadata about the input and variables through which it is accessed.

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 metadata 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 metadata 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: Client/server news reader app 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 (iPhone, 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 Phone, respectively. The cross-compilation could be done in a way similar to Links [25], 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 demands. 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 demands using the type system. To make this practical, the system should also provide at least a partial support for automatic type inference, as the information about context demands makes the types more complex. An inspiring example might be the F# support for units of measure [53] – 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 the mouse pointer over an identifier. For fetchLocalNews, the tip could appear as follows:

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

Here, we use the notation $\tau_1 @ c \rightarrow \tau_2$ to denote a function that takes an input of type τ_1, produces a result of type τ_2 and has additional context demands 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 is just for illustration and 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 pieces 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 a 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 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 (with shared functions reused) – 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 string, depending on the types of the variables. For example, the following is a valid typing judgement in F# [108]:

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

This judgement assumes that the type of x and y is int and so the result must also be int. The expression might also be typeable in a context $x : \mathsf{string},\ y : \mathsf{string}$, but not in a context where types of x and y do not match.

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 send(read()) are {console, 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. This intuitive distinction is made more precise in Section 5.6.

TRACKING CONTEXT DEMANDS. The systems for tracking of context demands 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:

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

The expression queries a database and gets back a list of news based on the spefieid `time`. There are three pieces of 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. Third, the expression uses a special construct `time`, which requires access to *system clock*. The *coeffect systems* developed in this thesis capture this information in the following way (we slightly refine the notation later):

$$(\mathsf{password}{:}\mathsf{string}\ @\ \mathsf{sensitive}, \mathsf{cmd}{:}\mathsf{string})\ @\ \{\mathsf{database}, \mathsf{clock}\}$$
$$\vdash \mathsf{query}(\mathsf{cmd}, \mathsf{password}, \mathtt{time}) : \mathsf{news\ list}$$

The typing judgement includes an additional annotation that now captures contextual requirements of the expression. The annotation is attched to the variable context Γ. As discussed later (Section 4.2.3), this better reflects how contextual information are used in the type system, but it also matches the semantics (Section 5.2). Rather than attaching additional information to the *resulting type*, we attach them to the variable context Γ.

In other words, the context in coeffect systems consists of the available variables, but also tracks detailed information about the execution environment. In the above example, the system tracks metadata about the variables and annotates password as sensitive and it tracks requirements about the execution environment, for example, that the execution requires clock.

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 6.

CONTEXT DEMANDS AND LAMBDA ABSTRACTION. The difference between effects and coeffects becomes apparent when we consider lambda abstraction. Given an effectful expression such as `print "hi"`, the function `λx.print "hi"` is an effect free value that delays all effects. The message is printed when we run the function, but not when the function is declared.

Contextual properties do not follow this pattern. As discussed in Section 3.1.1, context demands cannot always be delayed. For example, consider a context-aware expression `time` that requires access to system clock. If we create this function on a server in a distributed system, send it to a client and then run it, the function body can access the clock in two ways. It can use the clock of the server (when the function was created), or it can use the clock of the client (when the function was executed). We return to this example in Section 3.2.2.

Another important difference between effects and coeffects becomes apparent when we consider their semantics. 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 throughout the thesis.

1.4 THESIS OUTLINE

This chapter shows why capturing how programs rely on the context, or execution environment, in which they execute 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 the theory of programming languages (including type and effect systems, categorical semantics and substructural logics).

- In Chapter 3, we present 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 6, we capture important contextual properties using a simple theoretical models. We develop the *flat coeffect calculus* that captures per-context properties and *structural coeffect calculus* that captures per-variable properties. We give a type system for the calculi and study their equational properties.

- In Chapter 5 we give categorical semantics of the flat coeffect calculus. This provides a unified way of defining the semantics of context-aware languages. We use the categorical semantics as a basis for *categorically-inspired translation* that turns context-aware programs into programs in a simple functional language. We prove that well-typed context-aware programs are translated to programs that "do not go wrong". The development is repeated for structural coeffects in Chapter 6.

- In Chapter 7, we use the translation as a basis for a prototype implementation of three simple context-aware programming languages. We use a format of web-based *interactive essay* (available at http://tomasp.net/coeffects), which shows how coeffects simplify programming with context by allowing the reader to write and run simple context-aware programs, but also explore the theory behind the implementation including the typing derivation and the translation.

- Related work is presented in Chapter 2 and, together with further work, throughout the thesis. Two important directions deserve further exploration. In Chapter 8, we outline a unified coeffect system that is capable of capturing both flat and structural properties. We also include a brief discussion of a different approach to tracking contextual information that arises from modal logics.

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. 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 (Section 2.1) by discussing how coeffects arise from the most common notion of context-dependence – variable binding. Next, we look at coeffects as the dual of effect systems (Section 2.2) and we extend the duality to category theory, looking at *comonads* (Section 2.3). We also consider type systems inspired by linear and bunched logic, which are closely related to our structural coeffects (Section 2.4). Finally, we also consider practical motivations for context-aware programming (Section 2.5).

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. We present the work through the coeffect view and so we sometimes focus on aspects that authors consider unimportant or we present the work differently than originally intended. When we do so, this is explicitly stated in the text.

2.1 COEFFECTS VIA STATIC AND DYNAMIC BINDING

Accessing a variable is arguably the simplest form of context-dependence, to the extent that we do not normally think of variables as a notion of context. However, variables fit well with our earlier description of context in programming: a block of code that accesses a variable can only be executed in an environment where the variable is available.

In this section, we look at variable binding through the perspective of context-requirements. We discuss ordinary variable binding and Haskell's implicit parameters [60], which provide an interesting point in the design space. Implicit parameters give an example of an ambiguity that arises more generally in context-aware programming, as well as one way of resolving it through *type-directed semantics*.

For a more general context-aware programming example, consider a program running on a laptop that accesses an implicit parameter representing a printer. When printing, the text may appear on my home printer (corresponding to static binding), or it may appear on the nearest printer based on my physical location (corresponding to dynamic binding).

2.1.1 *Variable binding*

Variable access represents a form of context-dependence. For example, an expression $x + y$ can be only evaluated if the environment provides values for variables x and y. A variable *requirement* can be satisfied in two standard

ways that are characterized as *dynamic* and *static* (or lexical) variable binding. Consider the following simple program:

```
let x = 10 in
let f = λy → x + y in
let x = 5 in
f 0
```

The program can be evaluated in two ways, depending on the variable binding mechanism:

- STATIC (LEXICAL) BINDING. In a language with static binding (such as ML or Java), the variable x inside the body of the lambda function is statically bound to the declaration in the lexical scope – that is, the variable on the second line – and the expression evaluates to 10.
- DYNAMIC BINDING. In a language with dynamic binding (some variants of LISP), the variable value is dynamically bound to the topmost value for the available kept on the stack during program execution. Thus, the x variable inside the lambda function refers to x defined on line 5 of the sample and the expression evaluates to 5.

When we view variable access as a context requirement, we can see that the body of the function $(x + y)$ requires a context that provides values for variables x and y. In static binding, the context demands of the body can be placed on the scope in which the function is defined (declaration site). In dynamic binding, the requirements are *delayed* and are placed on the scope in which the function is called (call site).

In static and dynamic scoping, all variable requirements are always placed on one site. However, those are not the only two options. It is conceivable that a language would use a mechanism that splits variable requirements differently and combines aspects of dynamic and static binding. For example, the language could use static binding by default, but resort to dynamic binding if a variable is not available in the lexical scope. One such system is implicit parameters discussed in the next section.

Languages with static scoping resolve variable bindings at compile-time. In contrast, languages with dynamic variable binding cannot resolve variables at compile-time. They typically perform runtime checks – if a program attempts to access a variable that is not available in the environment, a runtime error occurs. Implementing static checking for dynamic binding is also possible, but it requires a more sophisticated type system (Section 2.2.2), while implementing static binding *without* checking would be cumbersome and so dynamically scoped languages are often dynamically typed.

2.1.2 *Implicit parameter binding*

Haskell uses static binding for ordinary variables, but GHC additionally provides a feature named *implicit parameters* [60] that adds a special kind of variables, written as ?param, which use a particular combination of static and dynamic binding.

The following two examples are variations on the one discussed in Section 2.1.1, obtained by replacing a variable x with an implicit parameter ?x. On the left, the implicit parameter is declared both in the static scope and in the dynamic scope. On the right, the implicit parameter is available only in the dynamic scope:

```
let f =
    let ?x = 10 in
    λy → ?x + y in
let ?x = 5 in
f 0
```

```
let f =
    λy → ?x + y in
let ?x = 5 in
f 0
```

The binding rules for Haskell's implicit parameters can be summarized as "*static binding when possible, dynamic binding when needed*". If an implicit parameter is available in static scope, then the value is statically bound and the context requirement is satisfied using the declaration site context. Otherwise, the context requirement is delayed and has to be satisfied at the call site. In the example on the left, ?x is bound to 10 and so the function f has no delayed context demands and thus the expression evaluates to 10. On the right, the context demand for ?x is *delayed* and is satisfied via dynamic binding when calling the function. The expression evaluates to 5.

In Haskell, the type system checks that bindings for all required implicit parameters are available and so no runtime errors can occur. The type of the function f on the left is `int → int`, while the type of the f function on the right is {?x : `int`} ⇒ `int → int`. The part before ⇒ specifies the required implicit parameters that need to be available in the environment when calling the function. It is worth noting that the syntax is similar to the one used by type-class constraints. Those can be viewed as context demands too (Section 3.2.1).

THESIS PERSPECTIVE. The three different binding mechanisms discussed so far can be seen as different ways of splitting context demands of a particular kind into static and dynamic parts. Dynamic and static binding represent the opposite ends of the design spectrum and Haskell's implicit parameters are an interesting point inside the wider spectrum.

In this thesis, we consider various notions of context, using implicit parameters as one of several motivating examples. Implicit parameters are a particularly valuable example, because they clearly illustrate the ambiguity inherent in context-aware programs – the context demands of a function can be satisfied using the context available at declaration site or using the context available at the call site. Recall our earlier printer example – a language that provides access to resources in the context needs to provide enough flexibility in handling such ambiguities, be it implicit parameter values available in scope or printers available in the physical environment.

We aim to find a description of context-aware languages that does not make ad-hoc decisions about how context demands are split between the declaration site and the call site. While Haskell's solution for implicit parameters might be the most reasonable one for that particular case, this thesis argues that other notions of context require different domain-specific choices and the general framework of context-aware programming should make that possible.

2.1.3 *Resolving ambiguity*

In many practical programming languages, the value and semantics of an expression depends on its type derivation. In order to assign unique semantics to an expression, the choice is typically hidden behind a mechanism that selects one preferred type derivation.

This mechanism serves as an inspiration for our approach to resolving ambiguity inherent in context-aware programs. This section discusses a brief example using the F# language [108], before revisiting the implicit parameters example.

Consider an F# lambda expression $(\lambda x \rightarrow x.\mathsf{Length})$, which takes an object x and returns the value of its Length property. F# is a nominally-typed language meaning that, in isolation, the function has an ambiguous meaning[1]. It can be a function taking an array, it can be a function taking a string, or it can be a function taking one of the other .NET types that are equipped with the Length property.

The semantics of the function depends on the typing derivation. For example, for arrays, it is compiled using the ldlen intermediate language (IL) instruction, while or strings, it is compiled using call instruction (calling the property getter). In F#, the compiler chooses an appropriate typing derivation. For example:

```
["hello";"world"] |> List.map (λs → s.Length)
[Array.empty; Array.create 100 0 ] |> List.map (λs → s.Length)
```

The $|>$ operator passes the value on the left to the function on the right. In the first case, the compiler infers that the type of the input is a list of strings and so the type of the lambda function becomes $\mathsf{string} \rightarrow \mathsf{int}$. In the second case, the list contains two arrays (an empty array and an array containing one hundred zero values) and so the type of the lambda function is $\mathsf{int}[] \rightarrow \mathsf{int}$. The important points about the example are:

1. The semantics of the function $(\lambda x \rightarrow x.\mathsf{Length})$ depends on its type. For arrays, it is compiled using a special IL instruction, while for strings, it calls a property getter.

2. The compiler chooses an appropriate typing derivation. In the above case, this is done based on the context in which the expression appears, but other options are possible (in some cases, there is a *default* resolution; often, the compiler requires an explicit typing annotation).

3. An expression without a type derivation does not have semantics. For example, given $\mathsf{List.map}\ (\lambda s \rightarrow s.\mathsf{Length})$, the F# compiler fails to infer a type; the expression is not well-typed and does not have a semantics.[2]

The function $\lambda y \rightarrow ?x + y$ in Haskell also has multiple possible typing derivations and its semantics varies depending on its type. If the lexical scope contains a binding for $?x$, the function type is $\mathtt{int} \rightarrow \mathtt{int}$ and it captures the value from the lexical scope. Otherwise, the type of the function is $\{?x : \mathtt{int}\} \Rightarrow \mathtt{int} \rightarrow \mathtt{int}$ and it reads the parameter value from a hidden dictionary that is passed together with the input from the call site.

THESIS PERSPECTIVE. Just like the F# function in the above example, certain expressions in context-aware languages developed in this thesis have multiple valid typing derivations and their semantics depends on the type. In F#, the compiler determines a unique typing derivation based on other parts of the program (if type is not uniquely determined, it either chooses

1 In contrast, in a structurally-typed language, the function would have a unique typing in isolation. In OCaml, the type would be $\langle \mathsf{Length} : \mathsf{'a} \rangle \rightarrow \mathsf{'a}$.

2 Alternatively, the compiler could choose default typing among multiple options. The F# compiler does this for the $+$ operator, which can be used on float and int types, but the compiler chooses int as the default.

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

$$(\textit{write})\ \frac{\Gamma \vdash e : \tau, r \qquad l : \mathsf{ref}_\rho\ \tau \in \Gamma}{\Gamma \vdash l \leftarrow e : \mathsf{unit}, r \cup \{\mathsf{write}(\rho)\}}$$

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

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

Figure 3: Simple effect system

a default or fails). In our languages, we also determine a unique typing derivation. However, rather than relying on type information from other parts of the program, we explicitly define an algorithm that chooses the preferred unique derivation (Section 4.3 and Section 6.3).

This approach decouples two important aspects of context-aware programming and lets us study them independently – the semantics of context-aware programs and the domain-specific way of resolving ambiguities in how context demands are satisfied. In our treatment of implicit parameters, we consider multiple typing derivations (representing a range with static and dynamic scoping at opposite ends), but we uniquely choose one preferred typing (mimicking the behaviour of GHC for implicit parameters).

2.2 COEFFECTS VIA TYPE AND EFFECT SYSTEMS

Introduced by Gifford and Lucassen [38, 62], 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 [106], communication in message-passing systems [51] and atomicity in concurrent applications [34].

Type and effect systems are usually specified as judgements of the form $\Gamma \vdash e : \tau, r$, meaning that the expression e has a type τ in a (free-variable) context Γ and additionally may have effects described by r. 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. Filinski [32] refers to this approach as *descriptive*[3].

2.2.1 *Simple effect system.*

The structure of a simple effect system[4] 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 ($\mathsf{ref}_\rho\ \tau$) 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

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

4 Most work on effect systems uses σ or F for effect annotations. We use letters r, s, t and also distinguish effect or coeffect annotations by colour.

way to build such over-approximation. The effects are represented as a set of effectful actions that an expression may perform and the effectful action (*write*) adds a primitive effect $\mathsf{write}(\rho)$.

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

2.2.2 *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 $\Gamma\,@\,r \vdash e : \tau$, associating the additional information with the context (left-hand side) of the judgement rather than with the result (right-hand side) as in $\Gamma \vdash e : \tau, r$. This change alone would not be very interesting – we simply used different syntax to write a predicate with four arguments. The more interesting difference is how the lambda abstraction rule looks.

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 demands, application (*app*) combines context demands of the two sub-expressions and latent context-requirements of the function. The (*abs*) rule is different than the corresponding rule for effect systems – the resource requirements of the body $r \cup s$ are split between the *immediate context-requirements* associated with the current context $\Gamma\,@\,r$ and the *latent context-requirements* of the function.

This is where context-aware languages permit multiple valid typing derivations as discussed in Section 2.1.3. In the example here, a 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 timezone, it can use the resource available on the client-side). In other words, resources in this example support both static (lexical) and dynamic scoping. Out of the multiple valid typing derivation, we would choose one – for example, capturing only those server-side resources that are not available on the client-side[5]. We discuss this system in detail in Section 3.2.1.

2.3 COEFFECTS VIA LANGUAGE SEMANTICS

Another pathway to coeffects leads through the semantics of effectful and context-dependent computations. In a pioneering work, Moggi [67] showed that effects (including partiality, exceptions, non-determinism and I/O) can be modelled using the category theoretic notion of *monad*.

When using monads, we distinguish effect-free values τ from programs, or computations $M\tau$. 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)* $M : \mathcal{C} \to \mathcal{C}$

5 This can be characterized as "*dynamic binding when possible, static binding when needed*" and it is, quite curiously, the opposite choice than the one used by Haskell's implicit parameters.

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

$$(\textit{access})\ \frac{\Gamma \mathbin{@} r \vdash e : \mathsf{res}_\rho\ \tau}{\Gamma \mathbin{@} r \cup \{\mathsf{access}(\rho)\} \vdash \mathtt{access}\ e : \tau}$$

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

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

Figure 4: Simple coeffect system

- 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 $f : \alpha \to M\beta$ *and* $g : \beta \to M\gamma$*:*

$$\mathsf{bind}\ \mathsf{unit} = \mathsf{id} \qquad (\textit{left identity})$$

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

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

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

2.3.1 *Effectful languages and meta-languages*

Moggi uses monads to define two formal systems. In the first formal system, a monad is used to model the *language* itself. This means that the semantics of a language is given in terms of a one specific monad and the semantics can be used to reason about programs in that language. To quote *"When reasoning about programs one has only one monad, because the programming language is fixed, and the main aim is to prove properties of programs"* [67, 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"* [67, 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 (the alternative is discussed in Section 8.2). To clarify the difference, the following two sections show a minimal example of both formal systems. We follow Moggi and start with language where judgements have the form $x{:}\tau_1 \vdash e : \tau_2$ with exactly one variable[6].

6 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.

LANGUAGE SEMANTICS. When using monads to provide semantics of a language, we do not need to extend the language in any way – we assume that the language already contains the effectful primitives (such as the assignment operator $x \leftarrow e$). A judgement of the form $x : \tau_1 \vdash e : \tau_2$ is interpreted as a morphism $\tau_1 \to M\tau_2$, meaning that any expression is interpreted as an effectful computation. The semantics of variable access and the application of a primitive function f is interpreted as follows:

$$\begin{aligned} [\![x{:}\tau_1 \vdash x : \tau_1]\!] &= \mathsf{unit}_M \\ [\![x{:}\tau_1 \vdash f\ e : \tau_3]\!] &= (\mathsf{bind}_M\ f) \circ [\![e]\!] \end{aligned}$$

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 τ_2 and that f is a (primitive) function $\tau_2 \to M\tau_3$. The semantics lifts the function f using bind_M to a function $M\tau_2 \to M\tau_3$ which is compatible with the interpretation of the expression e.

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

$$\begin{aligned} \tau &:= \mathsf{num} \mid \tau_1 \to \tau_2 \mid M\tau \\ e &:= x \mid f\ e \mid \mathtt{return}_M\ e \mid \mathtt{let}_M\ x \Leftarrow e_1\ \mathtt{in}\ e_2 \end{aligned}$$

The types consist of the primitive type, function type and a type constructor that represents monadic computations. Thus the expressions in the language can create both effect-free values, such as τ and computations $M\tau$. The additional expression $\mathtt{return}_M$ is used to create a monadic computation (with no effects) from a value and $\mathtt{let}_M$ sequences 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:

$$\begin{aligned} [\![x{:}\tau \vdash x : \tau]\!] &= \mathsf{id} \\ [\![x{:}\tau_1 \vdash f\ e : \tau_3]\!] &= f \circ [\![e]\!] \\ [\![x{:}\tau_1 \vdash \mathtt{return}_M\ e : M\tau_2]\!] &= \mathsf{unit}_M \circ [\![e]\!] \\ [\![x{:}\tau_1 \vdash \mathtt{let}_M\ y \Leftarrow e_1\ \mathtt{in}\ e_2 : M\tau_3]\!] &= \mathsf{bind}_M\ [\![e_2]\!] \circ [\![e_1]\!] \end{aligned}$$

In this system, the interpretation of variable access becomes a simple identity function and application is just composition. Monadic computations are constructed explicitly using $\mathtt{return}_M$ (interpreted as unit_M) and they are also sequenced explicitly using the $\mathtt{let}_M$ 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"*. 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 systems and semantics) rather than to build a general framework for reasoning about languages with contextual properties.

2.3.2 *Marriage of effects and monads*

The work on effect systems and monads both tackle the same problem – representing and tracking of computational effects. The two lines of research

have been joined by Wadler and Thiemann [125]. This requires extending the categorical structure. A monadic computation $\tau_1 \to M\tau_2$ means that the computation has *some* effects while the judgement $x : \tau_1 \vdash e : \tau_2, r$ specifies *what* effects the computation has.

To solve this mismatch, Wadler and Thiemann use a *family* of monads $M^r\tau$ with an annotation that specifies the effects that may be performed by the computation. In their system, an effectful function $\tau_1 \xrightarrow{r} \tau_2$ is modelled as a pure function returning monadic computation $\tau_1 \to M^r\tau_2$. Similarly, the semantics of a judgement $x : \tau_1 \vdash e : \tau_2, r$ can be given as a function $\tau_1 \to M^r\tau_2$. The precise nature of the family of monads has been later called *indexed monads* by Tate [107] and further developed by Atkey [6] in his work on *parameterized monads* and Katsumata [52].

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.

2.3.3 *Context-dependent languages and meta-languages*

The theoretical parts of this thesis extend the work of Uustalu and Vene who use comonads to give the semantics of dataflow computations [115] and more generally, notions of *context-dependent computations* [114]. 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* [2] in more details.

The utility of comonads has been explored by a number of authors before. Brookes and Geva [16] use *computational* comonads for intensional semantics[7]. In functional programming, Kieburtz [55] 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 [11], linking programming languages derived from modal logics to comonads. One such language has been reconstructed by Pfenning and Davies [87]. Nanevski et al. extend this work to Contextual Modal Type Theory (CMTT) [70], which again shows the importance of comonads for *context-dependent* computations.

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

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

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

7 The structure of a computational comonad has been also used by the author of this thesis to abstract evaluation order of monadic computations [81].

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

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

The definition is dual to a monad. Intuitively, the counit operation extracts a value α from a value that carries additional context $C\alpha$. The cobind operation turns a context-dependent function $C\alpha \to \beta$ into a function that takes a value with context, applies the context-dependent function to value(s) in the context and then propagates the context. The next section makes this intuitive definition more concrete. More detailed discussion about comonads can be found in Orchard's PhD thesis [75].

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

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

The counit operation returns the current (first) element of the (non-empty) list. The cobind operation creates a new list by applying the context-dependent function f to the entire list, to the suffix of the list, to the suffix of the suffix and so on. Interestingly, it preserves the *shape* of the list as it turns a list of n elements into another list of n elements.

In causal dataflow, 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 dataflow:

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

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

Now, consider a simple dataflow 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 τ, the expression `prev` e has also type τ. The fact that the language models dataflow and values are lists (of past values) is a matter of semantics, which is defined as follows:

$$\begin{aligned} [\![x{:}\tau \vdash x : \tau]\!] &= \mathsf{counit}_C \\ [\![x{:}\tau_1 \vdash f\ e : \tau_3]\!] &= f \circ (\mathsf{cobind}_C\ [\![e]\!]) \\ [\![x{:}\tau_1 \vdash \texttt{prev}\ e : \tau_2]\!] &= \mathsf{prev} \circ (\mathsf{cobind}_C\ [\![e]\!]) \end{aligned}$$

The semantics follows that of effectful computations using monads. A variable access is interpreted using counit_C (extract the variable value); compo-

sition 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 $x : \tau \vdash \texttt{prev}\ (\texttt{prev}\ x) : \tau$ 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 a *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. Note that this is what happens in the unusual (*abs*) rule in Figure 4, which distinguishes coeffect systems from effect systems.

We return to this topic when discussing lambda abstraction in Section 3.1.1 and semantics of flat coeffect systems in Section 5.2.

META-LANGUAGE INTERPRETATION. To demonstrate the approach that employs comonads as part of a meta-language, we look at an example inspired by the work of Pfenning et al. [87, 70]. We do not attempt to provide a precise overview of their work. The main purpose of the following discussion is to provide a different intuition behind comonads, and to present an example of a language that includes comonad as a type constructor, together with language primitives corresponding to comonadic operations[8].

In languages inspired by modal logics, types can have the form $\Box\tau$. In the work of Pfenning and Davies, this is the type of a term that is provable with no assumptions. In the ML5 language by Murphy et al. [68, 69], the $\Box\tau$ type means *mobile code*, that is code that can be evaluated at any node of a distributed system (the evaluation corresponds to the axiom $\Box\tau \to \tau$). Finally, Davies and Pfenning [29] consider staged computations and interpret $\Box\tau$ as a type of unevaluated expressions of type τ (with no free variables).

In Contextual Modal Type Theory, the modality $\Box$ is further annotated with the free variables of the (unevaluated) expression. We write $\Box^{\Psi}\tau$ for a type of expressions that requires a context Ψ. The type is a comonadic counterpart to *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 $C^{\Psi}\tau$ 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 indicates that we can evaluate a closed (unevaluated) term and obtain a value. Interestingly, the rule requires a specific context annotation (empty set of free variables). It is not possible to evaluate an open term.
- (*letbox*) corresponds to $\mathsf{cobind} : (C^{\Psi}\alpha \to \beta) \to C^{\Psi,\Phi}\alpha \to C^{\Phi}\beta$. 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.

8 In fact, Pfenning et al. never mention comonads explicitly. This is done in later work by Gabbay et al. [36], but the connection between the language and comonads is not as direct as in case of monadic or comonadic semantics covered in the previous section.

$$\text{(eval)}\ \frac{\Gamma \vdash e : \Box^{\emptyset}\tau}{\Gamma \vdash !e : \tau}$$

$$\text{(letbox)}\ \frac{\Gamma \vdash e_1 : \Box^{\Phi,\Psi}\tau_1 \qquad \Gamma, x{:}\Box^{\Phi}\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let box}\ x = e_1\ \texttt{in}\ e_2 : \Box^{\Psi}\tau_2}$$

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

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*). 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. We return back to the meta-language approach of embedding comonads in Section 8.2.

THESIS PERSPECTIVE. As mentioned earlier, we are interested in designing context-dependent languages and so we use comonads for *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 demands more precisely (by annotating the types) and we add indexing to comonads and link the two by giving a formal semantics. The indexing allows us to capture applications that do not fit into the model provided by plain comonads.

The *meta-language* approach of Pfenning et al. is closely related to our work. Most importantly, Contextual Modal Type Theory (CMTT) uses indexed $\Box$ modality which corresponds 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. [36], but the meta-language employed by CMTT does not directly correspond to comonadic operations. For example, our (*letbox*) 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.

2.4 COEFFECTS VIA SUBSTRUCTURAL 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: $\Gamma\,@\,r \vdash e : \tau$. 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 is added to the variable context (as a product), 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.

$$(\textit{exchange})\quad \frac{\Gamma, x{:}\tau_1, y{:}\tau_2 \vdash e : \gamma}{\Gamma, y{:}\tau_2, x{:}\tau_1 \vdash e : \gamma}$$

$$(\textit{weakening})\quad \frac{\Gamma, \Delta \vdash e : \gamma}{\Gamma, x{:}\tau, \Delta \vdash e : \gamma}$$

$$(\textit{contraction})\quad \frac{\Gamma, x{:}\tau_1, y{:}\tau_1, \Delta \vdash e : \tau_2}{\Gamma, x{:}\tau_1, \Delta \vdash e[y \leftarrow x] : \tau_2}$$

Figure 6: Exchange, weakening and contraction typing rules

In this section, we look at type systems that reconsider Γ in a number of ways. First of all, substructural type systems [126] restrict the use of variables in the language. Most famously linear type systems introduced by Wadler [123] can guarantee that a variable is used exactly once. This has interesting implications for memory management and I/O.

In bunched typing developed by O'Hearn [73], the variable context is a tree formed by multiple different constructors (e.g. one that allows sharing and one that does not). Most famously, bunched typing has contributed to the development of separation logic [74] (starting a fruitful line of research in software verification), but it is also interesting on its own.

2.4.1 *Substructural type systems.*

Traditionally, Γ is viewed as a set of assumptions and typing rules admit (or explicitly include) three transformations that manipulate the variable contexts which are shown in Figure 6. The (*exchange*) rule allows reordering of 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 (in the rule, this is done explicitly by joining multiple variables into a single one using substitution).

In substructural type systems, the assumptions are typically treated as a list. As a result, they have to be manipulated explicitly. Different systems allow different subsets 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. Substructural type systems provide a way to do this. Even if we allow all three operations, we can use a variation on the three rules (exchange, weakening and contraction) to track which variables are used and how (and to track additional contextual information about variables).

2.4.2 *Bunched type systems.*

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

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

$$(\textit{exchange1})\ \frac{\Gamma(\Delta, \Sigma) \vdash e : \alpha}{\Gamma(\Sigma, \Delta) \vdash e : \alpha}$$

$$(\textit{exchange2})\ \frac{\Gamma(\Delta; \Sigma) \vdash e : \alpha}{\Gamma(\Sigma; \Delta) \vdash e : \alpha}$$

$$(\textit{weakening})\ \frac{\Gamma(\Delta) \vdash e : \alpha}{\Gamma(\Delta; \Sigma) \vdash e : \alpha}$$

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

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

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

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

One important note about bunched typing is that it requires a different interpretation. The omission of weakening and contraction in linear logic means that variable must 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 $\tau_1 \to \tau_2$ and $\tau_1 \mathrel{-\!\!*} \tau_2$ (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 related to coeffects – for more information, refer to O'Hearn's introduction [73].

THESIS PERSPECTIVE. From the perspective of substructural and bunched types, 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 6, we could define contexts as follows:

$$\Gamma, \Delta, \Sigma := x{:}\alpha \mid 1 \mid \Gamma, \Gamma \mid \Gamma \mathbin{@} r$$

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 $\Gamma r \vdash e : \tau$). We can also annotate individual components. For example, a context containing variables x, y, z where only x is used could be written as $(x{:}\tau_1) \mathbin{@} \mathsf{used}, (y{:}\tau_2, z{:}\tau_3) \mathbin{@} \mathsf{unused}$.

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.

2.5 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 [97, 61] (which also inspired the motivating 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 [48]. The COP paradigm has been later implemented by programming language features. Costanza [26] develops a domain-specific LISP-like language ContextL and Bardram [7] 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 [9, 31]. 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, focusing on type systems as the primary way of capturing contextual properties.

2.6 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 primary goal of the chapter. The primary goal was to present the idea of coeffects as a logical follow up to a number of research directions. For this reason, we highlighted only certain aspects of the discussed related work – the remaining aspects as well as important technical details are covered throughout the thesis.

The first pathway follows as a generalization of static and dynamic variable binding. Variable binding can be seen as the most primitive form of context-dependence and coeffects provide a generalization that can capture different binding mechanisms in a unified way. In the second pathway, we looked at the dual of well-known work on effect systems. However, this is not simply a syntactic transformation. As we further discuss in the next chapter, 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 substructural type systems. Coeffect systems naturally arise by annotating bunches in bunched logics with additional information. In this thesis, we mostly follow the first two approaches.

3 CONTEXT-AWARE SYSTEMS

Software developers as well as programming language researchers choose abstractions based not just on how appropriate they are. Other factors 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] that the researcher unconsciously subscribes to.

For tracking of effects, such *standard tools* are well known. When faced with an effectful computation, programming language designers immediately pick monads. For context-aware computations, there are no standard tools. Thus contextual properties may, at first, appear as a set of disconnected examples. Existing systems that capture contextual properties use a wide range of methods including special-purpose type systems, approaches arising from modal logic S4, as well as techniques based on abstractions designed for other purpose, most frequently monads.

This chapter reviews some of the existing context-aware programming abstractions and presents them in a uniform way. We start with disconnected examples, but at the end, we will see that they share a common pattern[2].

CHAPTER STRUCTURE AND CONTRIBUTIONS

- We characterize contextual properties – Section 3.1 explains what is a *coeffect* and contrasts it with a better known notion of *effect*. It explains what is the nature of properties that can be tracked using coeffect systems presented in this thesis.
- We describe a number of simple calculi for tracking a wide range of contextual properties. The systems are adapted from diverse sources (type systems, static analyses, logics) and apply to various domains (cross-compilation, liveness, distributed computing, dataflow, security), but share a common structure.
- The uniform presentation of the systems is the key contribution of this chapter. We distinguish between *flat coeffect* systems (Section 3.2) and *structural coeffect* systems (Section 3.3). This common structure is precisely captured by the two *coeffect calculi* in the upcoming chapters.
- In addition, the coeffect systems for tracking the number of accessed past values in dataflow languages (Sections 3.2.4 and 3.3.3) presents novel results and can be used to optimize dataflow programs.

3.1 STRUCTURE OF COEFFECT SYSTEMS

When introducing coeffect systems in Section 2.2.2, we related coeffect systems with effect systems. Effect systems track how a program affects the environment, or, in other words capture some *output impurity*. In contrast, coeffect systems track what a program requires from the execution environment, or *input impurity*.

1 A research programme, as introduced by Lakatos [58], is a network of scientists sharing the same basic assumptions and techniques.

2 The different properties captured by monads may appear similarly disconnected at first!

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

$$(\textit{effect})\quad \frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2 \,\&\, \sigma}{\Gamma \vdash \lambda x.e : \tau_1 \xrightarrow{\sigma} \tau_2 \,\&\, \emptyset}$$

Figure 8: Lambda abstraction for pure and effectful computations

Effect systems generally use judgements of the form $\Gamma \vdash e : \tau \,\&\, \sigma$, associating effects σ with the output type. We write coeffect systems using judgements of the form $\Gamma \circledast \sigma \vdash e : \tau$, associating the context demands with Γ. Thus, we extend the traditional notion of free-variable context Γ with richer notions of context. This notation emphasizes the right intuition, but there are more important differences between effects and coeffects.

3.1.1 *Effectful lambda abstraction*

As outlined in Section 1.3, the difference between effects and coeffects becomes apparent when we consider lambda abstraction. The typical lambda abstraction rule for effect systems looks as (*effect*) in Figure 8. Wadler and Thiemann [125] 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.*

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 [107] 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. We could track properties associated with values in data structures (e. g. fields of a tuple), but this is left as future work.

VARIABLE COEFFECTS. As discussed in Section 2.1, variable access can be seen as a basic form of context requirement in λ-calculus. The expression x is typeable only in a context that contains $x : \tau$ for some type τ.

In lexically scoped languages, lambda abstraction (*pure*), as shown in Figure 8, splits the free-variable context of an expression into two parts. At runtime, 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). In the type checking, the splitting is determined syntactically. The notation $\lambda x.e$ *names* the variable whose value comes from the call site.

Flat and structural coeffects also split context-requirements between the declaration site and the call site. The flat and structural coeffects capture two different ways of doing this.

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 demands 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 variable context of such expression will be annotated with a set of required resources, i. e. $\Gamma\,@\,\{\ \mathsf{gps},\ \mathsf{time}\ \}$.

The interesting case is when we construct a lambda function $\lambda x.e$, marshall it and send it to another node. In systems such as Acute [98], 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 transfer 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 *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 detail 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, there is no syntactic structure that determines how the requirements are split. As mentioned in Section 2.1.3, we decouple the definition of semantics from the domain-specific choice that determines how context demands are satisfied. We capture the choice in the type and give semantics over a *typing derivation*. A domain-specific algorithm then chooses the desirable typing – for example, by preferring resources available on the client over resources available on the server.

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 linked to variables in programs. Structural coeffects follow the lexical scoping structure determined by the typing rules.

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, relative to the current cursor, that may be accessed. For example, consider an expression $x[\mathtt{cursor}] = y[\mathtt{cursor} - 1] + y[\mathtt{cursor} + 1]$.

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 `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. The context annotation is written as: $x{:}\mathsf{Arr}, y{:}\mathsf{Arr} \mathbin{@} \langle (0,0), (-1,1) \rangle$.

Note that we attach the per-variable information as a *vector* of annotations associated with a *vector* of variables, which makes it possible to treat flat and structural coeffects uniformly (Section 6.6 and 5.2) and unify them into a single system (Section 8.1), which is capable of tracking both flat per-context coeffects and structural per-variable coeffects. Attaching structural coeffect annotations directly to individual variables would simplify some aspects of our formalism; we choose not to follow that approach as we envision that a fully unified theory of coeffects (which we attempt to approach in this thesis) would be able to capture variables, flat and structural coeffects using just a single mechanism.

In structural systems, the splitting of context is determined by the name (variable) binding. For example, consider a function that takes y and contains the above body: $\lambda y.x[\texttt{cursor}] = y[\texttt{cursor} - 1] + y[\texttt{cursor} + 1]$. 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 ambiguity arising from the splitting of requirements in flat coeffect systems.

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 n free variables of types τ_i can be modelled by a function $\tau_1 \times \ldots \times \tau_n \to \tau$ with a product on the left, but a single value on the right. In both effect systems and coeffect systems, we write the annotation as part of the function arrow. However, in the underlying categorical model, effects are attached to the result τ, while coeffects are attached to the context $\tau_1 \times \ldots \times \tau_n$.

Structural coeffects have one annotation per 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 coeffects are not trivially dual to effects.

It is useful to clarify how vectors are used in this thesis. Suppose we have a set $\mathcal{C}$ of *scalars* ranged over by r, s, t. 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 normal face r, s, t for scalars. We also say that a *shape* of a vector $\mathit{len}(\mathbf{r})$ (or more generally any container) determines the set of *positions* in a vector. So, a vector of a shape (length) n has positions $\{1, 2, \ldots, n\}$. We discuss containers and shapes further in Section 8.1 and also discuss how our use relates to containers of Abbott, Altenkirch and Ghani [2].

Just as in the usual pointwise multiplication of a vector by a scalar, we lift any binary operation on scalars into a scalar-vector one. For a binary operation on scalars $\circ : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$, we define $s \circ \mathbf{r} = \langle s \circ r_1, \ldots, s \circ r_n \rangle$ and $\mathbf{r} \circ s = \langle r_1 \circ s, \ldots, r_n \circ s \rangle$. 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 vari-

able contexts. Given vectors $\mathbf{r}, \mathbf{s}$ with (possibly different) shapes $\{1, \ldots, n\}$ and $\{1, \ldots, m\}$, the associative operation for concatenation $+\!\!+$ is defined as $\mathbf{r} +\!\!+ \mathbf{s} = \langle r_1, \ldots, r_n, s_1, \ldots, s_m \rangle$.

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 $\Gamma_1 +\!\!+ \langle x{:}\tau \rangle +\!\!+ \Gamma_2$.

3.2 FLAT COEFFECT SYSTEMS

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

As mentioned in the introduction, coeffect systems in this chapter may appear as a disconnected set of examples at first. Indeed, this section covers a diverse set of calculi including Haskell's implicit parameters (Section 3.2.1), distributed computing and cross-compilation (Section 3.2.2), liveness analysis (Section 3.2.3) and dataflow (Section 3.2.4).

For three of the examples, we present a type system and a simple semantics (given inductively over the typing derivation). We informally discuss how preferred typing derivation is chosen (to resolve the inherent ambiguity), but leave details to later chapters. Although the examples are not new, our novel presentation of the systems (and the fact that they appear side-by-side) makes it possible to see that they share a common structure. The structure is captured by a unified *flat coeffect calculus* in Chapter 4.

3.2.1 *Implicit parameters and type classes*

Haskell provides two examples of flat coeffects – type class constraints and implicit parameter constraints [124, 60]. 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 $\Gamma \mathbin{@} C \vdash e : \tau$.

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 not give a full type system.

IMPLICIT PARAMETERS. As discussed in Section 2.1.2, implicit parameters are a special kind of variables that support dynamic scoping. They make it possible 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 is defined, then the function will not require a value from the caller.

A simple language with support for implicit parameters has an expression `?param` to read a parameter and an expression[3] `letdyn ?param` $= e_1$ `in` e_2

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

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}. 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 by r. We also assume that implicit parameters are of type num, so the annotation can be a simple set of names (rather than mapping from names to types). In a purely dynamically scoped system, implicit parameters 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:

$$\mathsf{string} \xrightarrow{\{?\mathsf{width}\}} (\mathsf{string} \xrightarrow{\{?\mathsf{width},?\mathsf{size}\}} \mathsf{string}) \qquad (\textit{dynamic})$$

$$\mathsf{string} \xrightarrow{\{?\mathsf{width}\}} (\mathsf{string} \xrightarrow{\{?\mathsf{size}\}} \mathsf{string}) \qquad (\textit{mixed})$$

This is not a complete list of possible typings, but it demonstrates the options. The (*dynamic*) case requires the parameter ?width twice (the caller may provide different value, in which case, the semantics needs to specify which value is preferred). In the (*mixed*) case, the nested function captures the ?width parameter available from the declaration site. Using the latter typing, the 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 programs have multiple typing derivations and the semantics depends on the domain-specific choice of preferred 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 $\Gamma \mathbin{@} r \vdash e : \tau$ 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 names, i. e. $r, s, t \subseteq \mathsf{Names}$. The expressions include ?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 demands is (*param*). Accessing a parameter ?param requires it to be available in the context. The context

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

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

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

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

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

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

$$(letdyn)\ \frac{\Gamma\,@\,r \vdash e_1 : \tau_1 \qquad \Gamma\,@\,s \vdash e_2 : \tau_2}{\Gamma\,@\,r \cup (s \setminus \{?p : \tau_1\}) \vdash \mathtt{letdyn}\ ?p = e_1\ \mathtt{in}\ e_2 : \tau_2}$$

Figure 9: Coeffect rules for tracking implicit parameters

may provide more (unused) implicit parameters thanks to the subcoeffecting rule (*sub*).

When we read the rules from the top to the bottom, application (*app*) and let binding (*let*) simply union the context demands 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 ($\tau_1 \xrightarrow{s} \tau_2$) and thus an expression may have multiple valid typing derivations. Finally, (*letdyn*) removes the bound parameter from the set of requirements.

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 $?a + ?b$. 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 A.1:

$$\Gamma\,@\,\{?a : \mathsf{int}\} \vdash \lambda x.?a + ?b : \mathsf{int} \xrightarrow{\{?b:\mathsf{int}\}} \mathsf{int} \quad (1)$$

$$\Gamma\,@\,\{?b : \mathsf{int}\} \vdash \lambda x.?a + ?b : \mathsf{int} \xrightarrow{\{?a:\mathsf{int}\}} \mathsf{int} \quad (2)$$

$$\Gamma\,@\,\{?a : \mathsf{int}\} \vdash \lambda x.?a + ?b : \mathsf{int} \xrightarrow{\{?a:\mathsf{int},?b:\mathsf{int}\}} \mathsf{int} \quad (3)$$

$$\Gamma\,@\,\emptyset \vdash \lambda x.?a + ?b : \mathsf{int} \xrightarrow{\{?a:\mathsf{int},?b:\mathsf{int}\}} \mathsf{int} \quad (4)$$

The first two examples demonstrate that 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.

The next typing derivation (3) requires the parameter $?a$ from both the declaration site and the call site. This means that, at runtime, two values will be available. Our semantics for the system describes *dynamic rebinding*, meaning that when the caller provides a value for a parameter that is already specified by the declaration site, the new value hides the old one. This means that only the value from the call site is actually used. This (4) gives a more precise typing for this situation.

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

$$\frac{}{[\![\Gamma \mathbin{@} r \vdash ?p : \mathsf{num}]\!] = \lambda(\_,f).f\ ?p} \quad (\textit{param})$$

$$\frac{[\![\Gamma \mathbin{@} r' \vdash e : \tau]\!] = f}{[\![\Gamma \mathbin{@} r \vdash e : \tau]\!] = \lambda(x,g).f\ (x, g|_{r'})} \quad (\textit{sub})$$

$$\frac{[\![\Gamma, y : \tau_1 \mathbin{@} r \cup s \vdash e : \tau_2]\!] = f}{[\![\Gamma \mathbin{@} r \vdash \lambda y.e : \tau_1 \xrightarrow{s} \tau_2]\!] = \lambda((x_1,\ldots,x_n),g_1).\lambda(y,g_2).\ f\ ((x_1,\ldots,x_n,y),g_1 \uplus g_2)} \quad (\textit{abs})$$

$$\frac{\begin{array}{rcl}[\![\Gamma \mathbin{@} r \vdash e_1 : \tau_1 \xrightarrow{t} \tau_2]\!] &=& f_1\\ [\![\Gamma \mathbin{@} s \vdash e_2 : \tau_1]\!] &=& f_2\end{array}}{[\![\Gamma \mathbin{@} r \cup s \cup t \vdash e_1\ e_2 : \tau_2]\!] = \lambda(x,g).\ (f_1\ (x, g|_r))\ (f_2\ (x, g|_s), g|_t)} \quad (\textit{app})$$

$$\frac{\begin{array}{rcl}[\![\Gamma \mathbin{@} r \vdash e_1 : \mathsf{num}]\!] &=& f_1\\ [\![\Gamma \mathbin{@} s \vdash e_2 : \tau_2]\!] &=& f_2\end{array}}{\begin{array}{l}[\![\Gamma \mathbin{@} r \cup (s \setminus \{?p\}) \vdash\\ \quad \mathtt{letdyn}\ ?p = e_1\ \mathtt{in}\ e_2 : \tau_2]\!]\end{array} = \lambda(x,g).f_2\ (x, g|_{s\setminus\{?p\}} \uplus \{?p \mapsto (f_1\ (x, g|_r))\})} \quad (\textit{letdyn})$$

Assuming the following auxiliary definitions:

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

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 new `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 5.

Given an expression e of type τ that requires free variables Γ and implicit parameters r, the semantics is a function that takes a product of variables from Γ together with a dictionary of implicit parameters and returns τ:

$$[\![x_1 : \tau_1, \ldots, x_n : \tau_n \mathbin{@} r \vdash e : \tau]\!] \ :\ (\tau_1 \times \ldots \times \tau_n) \times (r \to \mathsf{num}) \to \tau$$

The dictionary is represented as a function from r to num. This means that it provides a num 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 the finite subset of names that are required according to the typing.

The dictionary is also attached to the inputs of all functions. That is, 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:

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

The definition of the semantics is shown in Figure 10. We use a notation that emphasizes the fact that the semantics is given over a typing derivation. On the left-hand side of $=$, we show the applied typing rule. The right-hand side of $=$ then shows the semantic functions assigned to the individual assumptions and the resulting semantics for the consequent.

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 g_1 and g_2 (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.

Finally, the (*letdyn*) rule specifies the semantics of the `letdyn` construct, which assigns a value to an implicit parameter. This is similar to (*app*), because it needs to evaluate the sub-expression first. After evaluating e_1, the result is added to the dictionary using $\uplus$. The semantics of ordinary let binding is omitted, because let binding can be treated as a syntactic sugar for $(\lambda x.e_2)\ e_1$.

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). This idea is captured more formally by the soundness proof for the operational semantics given in Chapter 5.

MONADIC SEMANTICS. Implicit parameters are related to the *reader monad*. The type $\tau_1 \times (r \to \mathsf{num}) \to \tau_2$ is equivalent to $\tau_1 \to ((r \to \mathsf{num}) \to \tau_2)$ through currying. Thus, we can express the function as $\tau_1 \to M\tau_2$ for $M\tau = (r \to \mathsf{num}) \to \tau$. 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.

When using the usual monadic semantics based on the reader monad, the semantics of the (*abs*) rule would be modified as follows:

$$\frac{[\![\Gamma, y : \tau_1 \mathbin{@} r \vdash e : \tau_2]\!]}{[\![\Gamma \mathbin{@} \emptyset \vdash \lambda y.e : \tau_1 \xrightarrow{t} \tau_2]\!]} = \frac{f}{\begin{array}{l}\lambda((x_1, \ldots, x_n), \_).\lambda(y, g).\\ \quad f\ ((x_1, \ldots, x_n, y), g)\end{array}}$$

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 monadic *unit*, which turns a function $\tau_1 \to M^r\tau_2$ into a monadic computation with no side-effects $M^\emptyset\tau_1 \to M^r\tau_2$.

As we discuss later in Section 5.6.3, 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 [124]. They provide a principled form of ad-hoc polymorphism (overloading). When 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:

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

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 twoTimes takes an additional 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 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: $\alpha \times \mathsf{Num}_\alpha \to \alpha$.

Type classes and implicit parameters show two important points about flat coeffect systems. First, the context demands 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. Here, the term "delay" refers to the fact that some implicit parameters may be captured (from the declaration site) at the time when the function is defined, but before it is executed. 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 means there the system does not have the pincipal type property, because the parameters can be split arbitrarily. 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. Both of these could be seen as an instance of implicit parameters, but we present them separately to illustrate other forms that coeffect systems can have. The examples given later (e. g. in Section 3.2.4) show coeffects that cannot be easily seen as implicit parameters.

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

// Searches database for a product; must 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

REBINDABLE RESOURCES. The need for parameters that support dynamic scoping also arises in distributed computing. To quote an example discussed by Bierman et al. [10]: *"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. If the resource is available on the target machine, the parameter may be *rebound* to the resource on the new machine. This is captured by the dynamic scoping rule. 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 the lexical scoping rule.

A practical language that supports rebindable resources is for example Acute [98]. In the following example, we use the construct `access Res` to represent access to a rebindable resource named Res. The following simple function accesses a database together with a current date; then it filters from the database based on the date:

```
let localNews = λ() →
  let db = access News in
  query db "SELECT * WHERE Location = %1" (access GPS)
```

When localNews 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 GPS, then GPS can be locally *rebound* to provide news for the reader's area. Otherwise, the default location needs to be obtained from the server.

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 requirements between declaration site (capturing remote reference) and call site (representing rebinding). For this reason, we do not discuss the system in detail and instead look at other uses.

CROSS-COMPILATION. A related issue with distributed programming is the need to target an 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 [119].

Links [25], F# WebTools and WebSharper [104, 80], ML5 and QWeSST [68, 94] and Hop [61] 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 poses 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 an 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 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 [25], functions can be annotated as client-side, server-side and database-side. F# WebTools [80] 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 [30] to cross-compile for multiple versions of the same platform.

TYPE SYSTEMS. Cross-compilation is similar to resource tracking (and thus to the tracking of implicit parameters), but it demonstrates a couple of new ideas that are important for flat coeffect systems. Unlike with implicit parameters, we will not give a full type system in this section, but we briefly look at two examples that 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.

Subcoeffecting and application are the same as the rules for implicit parameters. We just track supported environments using intersection as opposed to tracking required parameters using union. However, this symmetry does not hold for lambda abstraction (*abs*), which still uses *union*. This models the case when there are two ways of executing the function:

- The function is represented as executable code for a call site environment and is executed there, possibly after it is marshalled and transferred to another machine.

a.) Set-based type system for cross-compilation, inspired by Links [25]

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

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

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

b.) Version-based type system, inspired by Android API level [30]

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

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

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

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

- The function body is compiled for the declaration site environment; 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 (e. g. using explicit wrapping of unevaluated expressions) and the implementation also needs to be clarified. For a system that does this, see e. g. ML5 [68]). The key point of our brief example is that the algebraic structure of coeffect annotations may be more complex and use, for example, $\cap$ 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.

The ML5 language [68] mentioned above served as an inspiration for our example. It tracks execution environments using 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. However, it is based on the *meta-language* style of embedding modalities rather than on the *language-semantics* style (see Section 2.3.1). We return to this topic in Section 8.2.

3.2.3 *Liveness analysis*

Our next example shows the idea of coeffects from a different perspective. Rather than keeping additional information independent of the variable context, we track properties about how variables are used. Nevertheless, we still

look at the left-hand side of $\vdash$ and the structure of the typing rules and the semantics will be very similar.

Live variable analysis (LVA) [5] 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 [122] 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 an 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:

$$\texttt{let}\ \mathsf{constant42} = \lambda \mathsf{x} \to 42$$

$$\texttt{let}\ \mathsf{constant} = \lambda \mathsf{value} \to \lambda \mathsf{x} \to \mathsf{value}$$

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 the declaration site as live (this will be refined in a structural version).

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 variable uses to their declarations. The distinction between *must* and *may* is apparent when we look at an example with conditionals:

$$\begin{array}{l}\texttt{let}\ \mathsf{defaultArg} = \lambda \mathsf{cond} \to \lambda \mathsf{input} \to \\ \quad \texttt{if}\ \mathsf{cond}\ \texttt{then}\ 42\ \texttt{else}\ \mathsf{input}\end{array}$$

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* or *very busy* variable/expression).

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

TYPE SYSTEM. A type system that captures whole-context liveness annotates the context with value of a two-point lattice $\mathcal{L} = \{\mathsf{L}, \mathsf{D}\}$ where $\mathsf{L} \sqsubseteq \mathsf{D}$ (Figure 14 (a)). The annotation L marks the context as *live* and D stands for a *dead* context.

The typing rules for tracking whole-context liveness are shown in Figure 13. The language now includes numerical constants n. Accessing a constant (*num*) annotates the context as dead using D. This contrasts with variable access (*var*), which marks the context as live using L. A dead context

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

$$(\textit{num})\ \frac{}{\Gamma \mathbin{@} \mathsf{D} \vdash n : \mathsf{num}}$$

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

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

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

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

Figure 13: Coeffect rules for tracking whole-context liveness

(definitely not needed) can be treated as live context using the (*sub*) rule. This captures the *may* nature of the analysis.

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 a function obtained by sequential composition (first evaluate the argument e_2 and pass the result to the function value). 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 14 (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 14 (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. The (*let*) rule is again just a syntactic sugar for $(\lambda x.e_2)\ e_1$. This follows from the simple observation that $r \sqcup (s \sqcap r) = r$.

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

$$\begin{array}{llll} \mathsf{L} \sqcup \mathsf{L} = \mathsf{L} & \mathsf{L} \sqcup \mathsf{D} = \mathsf{D} & \mathsf{L} \sqcap \mathsf{L} = \mathsf{L} & \mathsf{L} \sqcap \mathsf{D} = \mathsf{L} \\ \mathsf{D} \sqcup \mathsf{L} = \mathsf{D} & \mathsf{D} \sqcup \mathsf{D} = \mathsf{D} & \mathsf{D} \sqcap \mathsf{L} = \mathsf{L} & \mathsf{D} \sqcap \mathsf{D} = \mathsf{D} \end{array}$$

b.) Sequential composition composes annotations using $\sqcup$:

$$f : \tau_1 \xrightarrow{r} \tau_2 \qquad g : \tau_2 \xrightarrow{s} \tau_3 \qquad g \circ f : \tau_1 \xrightarrow{r \sqcup s} \tau_3$$

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

c.) Pointwise composition composes annotations using $\sqcap$:

$$f : \tau_1 \xrightarrow{r} \tau_2 \qquad h : \tau_1 \xrightarrow{s} \tau_3 \qquad \langle f, h \rangle : \tau_1 \xrightarrow{r \sqcap s} \tau_2 \times \tau_3$$

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

Figure 14: Liveness annotations with sequential and pointwise composition

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 A.2:

$$\begin{array}{ll} (\lambda x.42)\ y & (1) \\ \mathsf{twoTimes}\ 42 & (2) \\ (\lambda x.x)\ 42 & (3) \end{array}$$

In the first case (1), the context is dead. 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. As showed in the examples, the type system for the liveness coeffect calculus marks the context of an expression $(\lambda x.42)\ y$ as dead. This means that the semantics of the above expression must not evaluate the argument y. In other words, the type system is only sound if the semantics includes dead code elimination.

To capture dead code elimination in the semantics, we add a special empty value and pass it as an argument to a function whose argument is not needed, so $(\lambda x.42)$ will be called with an empty value as argument (because it does not need its argument).

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 5.2.4. For now, it suffices to define the semantics using two separate rules:

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

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, 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 a constant using a global dictionary δ. In (*var*), the context is live and so the semantics performs a projection. Subcoeffecting 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 f_2 does not require its argument – x 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, lambda abstraction for liveness analysis has the principal types property. It simply duplicates the context demands. However, this still matches the property of coeffects that impurities cannot be delayed or thunked and attached just to the function arrow – we place requirements on both call site and declaration site.

The semantics of liveness reveals three interesting properties. Firstly, the coeffect calculus for liveness cannot be modelled as a monadic computation of the form $\tau_1 \to M\tau_2$. 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.

$$\frac{}{[\![\Gamma \mathbin{@} \mathsf{L} \vdash x_i : \tau_i]\!] = \lambda(x_1, \ldots, x_n).x_i} \quad (\textit{var})$$

$$\frac{}{[\![\Gamma \mathbin{@} \mathsf{D} \vdash n : \mathsf{num}]\!] = \lambda().n} \quad (\textit{num})$$

$$\frac{[\![\Gamma \mathbin{@} \mathsf{D} \vdash e : \tau]\!] = f}{[\![\Gamma \mathbin{@} \mathsf{L} \vdash e : \tau]\!] = \lambda x.f\ ()} \quad (\textit{sub-1})$$

$$\frac{[\![\Gamma \mathbin{@} r \vdash e : \tau]\!] = f}{[\![\Gamma \mathbin{@} r \vdash e : \tau]\!] = \lambda x.f\ x} \quad (\textit{sub-2})$$

$$\frac{[\![\Gamma, y : \tau_1 \mathbin{@} \mathsf{L} \vdash e : \tau_2]\!] = f}{[\![\Gamma \mathbin{@} \mathsf{L} \vdash \lambda y.e : \tau_1 \xrightarrow{\mathsf{L}} \tau_2]\!] = \lambda(x_1, \ldots, x_n).\lambda y.\ f\ (x_1, \ldots, x_n, y)} \quad (\textit{abs-1})$$

$$\frac{[\![\Gamma, y : \tau_1 \mathbin{@} \mathsf{D} \vdash e : \tau_2]\!] = f}{[\![\Gamma \mathbin{@} \mathsf{D} \vdash \lambda y.e : \tau_1 \xrightarrow{\mathsf{D}} \tau_2]\!] = \lambda().\lambda().f\ ()} \quad (\textit{abs-2})$$

$$\frac{\begin{array}{rcl}[\![\Gamma \mathbin{@} r \vdash e_1 : \tau_1 \xrightarrow{\mathsf{D}} \tau_2]\!] &=& f \\ [\![\Gamma \mathbin{@} r \vdash e_2 : \tau_1]\!] &=& \_\end{array}}{[\![\Gamma \mathbin{@} r \vdash e_1\ e_2 : \tau_2]\!] = \lambda x.(f\ x)\ ()} \quad (\textit{app-1})$$

$$\frac{\begin{array}{rcl}[\![\Gamma \mathbin{@} \mathsf{L} \vdash e_1 : \tau_1 \xrightarrow{\mathsf{L}} \tau_2]\!] &=& f_1 \\ [\![\Gamma \mathbin{@} \mathsf{D} \vdash e_2 : \tau_1]\!] &=& f_2\end{array}}{[\![\Gamma \mathbin{@} \mathsf{L} \vdash e_1\ e_2 : \tau_2]\!] = \lambda x.(f_1\ x)\ (f_2\ ())} \quad (\textit{app-2})$$

$$\frac{\begin{array}{rcl}[\![\Gamma \mathbin{@} r \vdash e_1 : \tau_1 \xrightarrow{\mathsf{t}} \tau_2]\!] &=& f_1 \\ [\![\Gamma \mathbin{@} s \vdash e_2 : \tau_1]\!] &=& f_2\end{array}}{[\![\Gamma \mathbin{@} r \sqcup (s \sqcap \mathsf{t}) \vdash e_1\ e_2 : \tau_2]\!] = \lambda x.(f_1\ x)\ (f_2\ x)} \quad (\textit{app-3})$$

Figure 15: Semantics that implements dead code elimination for λ-calculus

Finally, we discussed how the semantics of application arises from *sequential* and *pointwise* composition. This is an important aspect of coeffect systems – categorical semantics typically builds on *sequential* composition, but to model full λ calculus it needs more. For coeffects, we need *pointwise* composition where the same context is shared by multiple sub-expressions.

3.2.4 *Dataflow languages*

We used implicit parameters as our first example, because they show the simplest form of coeffects. Liveness requires a richer coeffect annotation structure, but the flat version is not practical. In this section, we look at a system with a structure similar to liveness that is not a toy example.

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 the 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, dataflow languages.

Lucid [121] is a declarative dataflow language designed by Wadge and Ashcroft. In Lucid, variables represent streams and programs are written as transformations over streams. A function application square(x) represents a stream of squares calculated from the stream of values x.

The dataflow approach has been successfully used in domains such as development of real-time embedded application where many *synchronous languages* [8] build on the dataflow paradigm. The following example is inspired by the Lustre [42] language and implements a 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):

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

When reading dataflow 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 [120] originally defined a monadic semantics, while Uustalu and Vene later used comonads [113]. In Chapter 4, we extend the latter approach. The present chapter presents a sketch of a concrete dataflow 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 dataflow 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. We can use a coeffect type system to track the maximal number of accessed past values. Here, the context is annotated with a single integer. The current value is 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; subcoeffecting (*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` (`prev` x) requires 2 past values.

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

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

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

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

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

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

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

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

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

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 $(\lambda x.e_2)\ e_1$.

ALGEBRA OF ANNOTATIONS. The coeffect annotations used in the liveness example in Section 3.2.3 form a two-point lattice. The annotations for dataflow do not form a lattice (as the absorption laws do not hold for addition and maximum), but the structure resembles that of *semiring*. As discussed later in Section 4.2.1, the coeffect annotations for many of our examples form a semiring, but this is not the case in general.

The coeffect annotations for dataflow forms a tropical semiring with $+$ as the multiplication, $\max$ as the addition, $-\infty$ as the zero element and 0 as unit. In this section, we do not use the $-\infty$ element, but this can be used to annotate unused variables as discussed in Section 4.2.4.

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 integers (starting from zero) and tick flips between 0 and 1. The full typing derivation is shown in Appendix A.3:

```
( 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 language discussed in this section is a *causal* dataflow 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 [114] model causal dataflow computations using a nonempty list $\mathsf{NeList}\ \tau = \tau \times (\mathsf{NeList}\ \tau + 1)$ over the input. A function $\tau_1 \to \tau_2$ is thus modelled as $\mathsf{NeList}\ \tau_1 \to \tau_2$. 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:

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

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 products*. To make the rules easier to follow, we write $\langle \mathbf{v}_1, \ldots, \mathbf{v}_n \rangle$ for an n-element list containing products. Products that model

$$\frac{}{[\![\Gamma @ 0 \vdash x_i : \tau_i]\!] = \lambda\langle(x_1,\ldots,x_n)\rangle.x_i} \quad (var)$$

$$\frac{[\![\Gamma @ n \vdash e : \tau]\!] = f}{[\![\Gamma @ n+1 \vdash \texttt{prev}\ e : \tau]\!] = \lambda\langle \mathbf{v}_0,\ldots,\mathbf{v}_{n+1}\rangle.\ f\ \langle \mathbf{v}_1,\ldots,\mathbf{v}_{n+1}\rangle} \quad (prev)$$

$$\frac{[\![\Gamma @ n' \vdash e : \tau]\!] = f}{[\![\Gamma @ n \vdash e : \tau]\!] = \lambda\langle \mathbf{v}_0,\ldots,\mathbf{v}_{n}\rangle.\ f\ \langle \mathbf{v}_0,\ldots,\mathbf{v}_{n'}\rangle} \quad (sub)$$

$$\frac{[\![\Gamma, y : \tau_1 @ n \vdash e : \tau_2]\!] = f}{[\![\Gamma @ n \vdash \lambda y.e : \tau_1 \xrightarrow{n} \tau_2]\!] = \lambda\langle \mathbf{v}_0,\ldots \mathbf{v}_n\rangle.\lambda\langle y_0,\ldots,y_n\rangle.\ f\ \langle(\mathbf{v}_0,y_0),\ldots,(\mathbf{v}_n,y_n)\rangle} \quad (abs)$$

$$\frac{[\![\Gamma @ r \vdash e_1 : \tau_1 \xrightarrow{t} \tau_2]\!] = f_1 \qquad [\![\Gamma @ s \vdash e_2 : \tau_1]\!] = f_2}{\Gamma @ \max(m, n+p) \vdash e_1\ e_2 : \tau_2 = \lambda\langle \mathbf{v}_0,\ldots,\mathbf{v}_{\max(m,n+p)}\rangle.\ (f_1\ \langle \mathbf{v}_0,\ldots,\mathbf{v}_m\rangle)\ \langle\ f_2\ \langle \mathbf{v}_0,\ldots,\mathbf{v}_n\rangle,\ldots,\ f_2\ \langle \mathbf{v}_p,\ldots,\mathbf{v}_{n+p}\rangle\ \rangle} \quad (app)$$

Figure 17: Semantics showing how past values are accessed in a dataflow language

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 (list of products) from the declaration site $\langle \mathbf{v}_0,\ldots,\mathbf{v}_n\rangle$ and one containing the argument (list of values) provided by the call site $\langle y_0,\ldots,y_n\rangle$. The semantics applies the well-known *zip* operation on the lists and passes the result to the body.

Finally, application (*app*) 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. Type systems have been used in the context of dataflow languages to check initialization properties [24] and resource-aware systems are capable of tracking how required values are accessed in dataflow programming [**?**]. This section formulates the problem in terms of coeffects.

The most interesting point about the dataflow system 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 dataflow examples, the context provides additional resources or values that may be accessed at runtime. However, coeffects can also track *permissions* or *capabilities* to perform some operation. We can invert the intuition behind liveness and use it 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 [33] as one, more advanced example. The calculus of capabilities of Cray et al. [27] is discussed later in Section 3.4.

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:

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

The declaration `newlock` creates a lock l protecting memory region ρ. We can than allocate mutable variables in that memory region (second line). An access to one or more mutable variables 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 by Flanagan and Abadi [33] 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:

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

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 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 future extension for coeffect systems.

3.3 STRUCTURAL COEFFECT SYSTEMS

In structural coeffect systems, the additional information is 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 three examples in detail – we revisit liveness and show a practically useful structural version of the system; we consider an example inspired by linear logic; finally, we revisit dataflow to get a more precise analysis. Although quite different, the common pattern among these three examples is somewhat easier to see, because they all track information about variable usage. We finish the section with a brief outline of several other applications.

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. Here, we revisit the problem and define a structural system that tracks liveness of individual variables.

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:

$$\begin{array}{l}\texttt{let constant} = \lambda y \to \lambda x \to y\\ \texttt{let answer} = (\lambda x \to x)\ 42\end{array}$$

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 substructural systems (discussed in Section 2.4) and add 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 Section 8.1, but the alternative is briefly explored in Section 8.2.

a.) Ordinary, syntax-driven rules along with subcoeffecting

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

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

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

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

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

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

b.) Structural rules for context manipulation

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

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

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

Figure 18: Structural coeffect liveness analysis

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 14.

The rules of the system are split into two groups. Figure 18 (a) shows the standard syntax-driven rules plus subcoeffecting. In (*var*), the context contains just the single accessed variable, which is annotated as live. Unused variables can be introduced using weakening. A constant (*const*) is accessed in an empty context, which also carries no annotations. The subcoeffecting rule (*sub*) uses a pointwise 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 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:

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

This definition is only for illustration and is revised in Chapter 6. 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 known from substructural type systems (Section 2.4). 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 $(\lambda x.v)\ y$. 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:

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

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:

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

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 constructor was applied to the entire context, i.e. $1 + (\tau_1 \times \ldots \times \tau_n)$. In the structural version, the semantics applies the option type constructor to 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:

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

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

Note that the product of the free variables is not a tuple of our language, but a special construction used only in the semantics. 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:

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

The rules that define the semantics are shown in Figure 19. To make the definition simpler, we treat the product $\tau'_1 \times \ldots \times \tau'_n$ as a flat list. Variables of product type are written in bold-face as $\mathbf{v}$ and individual values are written in normal face as x. An expression $(\mathbf{v}, x)$ should not be seen as a nested product, but simply as a product containing all variables from the product $\mathbf{v}$ together with one additional variable x at the end. We shall be more precise in Chapter 6.

In (*var*), the context contains just a single variable and so we do not even need to apply projection; (*const*) 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 subcoeffecting, which just turns some of the available values v_i to unit values $()$.

As dictated by the semantics, the application again needs to "implement" dead code elimination (otherwise the type system would be unsound). When the input parameter of the function f_1 is live (*app-1*), we first evaluate e_2 and then pass the result to f_1. 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 duplicate the value (no matter whether it is dead or live) and we use an auxiliary definition $x|_r$ to replace a live value with $()$ when only one of the contracted variables is live.

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 a 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 contexts.

a.) Semantics of ordinary expressions

$$\frac{}{[\![x{:}\tau \mathbin{@} \langle \mathsf{L} \rangle \vdash x : \tau]\!] \;=\; \lambda(x).x} \quad (\textit{var})$$

$$\frac{}{[\![() \mathbin{@} \langle\rangle \vdash n : \mathsf{num}]\!] \;=\; \lambda().n} \quad (\textit{num})$$

$$\frac{[\![\Gamma, y{:}\tau_1 \mathbin{@} \mathbf{r} \times \langle s \rangle \vdash e : \tau_2]\!] \;=\; f}{[\![\Gamma \mathbin{@} \mathbf{r} \vdash \lambda y.e : \tau_1 \xrightarrow{s} \tau_2]\!] \;=\; \lambda \mathbf{v}.\lambda y.f\ (\mathbf{v}, y)} \quad (\textit{abs})$$

$$\frac{\begin{array}{rcl}[\![\Gamma_1 \mathbin{@} \mathbf{r} \vdash e_1 : \tau_1 \xrightarrow{\mathsf{L}} \tau_2]\!] &=& f_1 \\ [\![\Gamma_2 \mathbin{@} \mathbf{s} \vdash e_2 : \tau_1]\!] &=& f_2\end{array}}{\begin{array}{rcl}[\![\Gamma_1, \Gamma_2 \mathbin{@} \mathbf{r} \times (\mathsf{L} \sqcup \mathbf{s}) \vdash e_1\ e_2 : \tau_2]\!] &=& \lambda(\mathbf{v_1}, \mathbf{v_2}).(f_1\ \mathbf{v_1})\ (f_2\ \mathbf{v_2})\end{array}} \quad (\textit{app-1})$$

$$\frac{\begin{array}{rcl}[\![\Gamma_1 \mathbin{@} \mathbf{r} \vdash e_1 : \tau_1 \xrightarrow{\mathsf{D}} \tau_2]\!] &=& f_1 \\ [\![\Gamma_2 \mathbin{@} \mathbf{s} \vdash e_2 : \tau_1]\!] &=& \_\end{array}}{\begin{array}{rcl}[\![\Gamma_1, \Gamma_2 \mathbin{@} \mathbf{r} \times (\mathsf{D} \sqcup \mathbf{s}) \vdash e_1\ e_2 : \tau_2]\!] &=& \lambda(\mathbf{v_1}, \mathbf{v_2}).(f_1\ \mathbf{v_1})\ ()\end{array}} \quad (\textit{app-2})$$

b.) Semantics of structural context manipulation
Using the auxiliary definition $x|_{\mathsf{L}} = x$ and $x|_{\mathsf{D}} = ()$:

$$\frac{[\![\Gamma \mathbin{@} \mathbf{r} \vdash e : \sigma]\!] \;=\; f}{[\![\Gamma, x{:}\tau \mathbin{@} \mathbf{r} \times \langle \mathsf{D} \rangle \vdash e : \sigma]\!] \;=\; \lambda(\mathbf{v}, ()).f\ \mathbf{v}} \quad (\textit{weak})$$

$$\frac{[\![\Gamma_1, x{:}\tau_1, y{:}\tau_2, \Gamma_2 \mathbin{@} \mathbf{r} \mathbin{\#} \langle s, t \rangle \mathbin{\#} \mathbf{q} \vdash e : \tau]\!] \;=\; f}{[\![\Gamma_1, y{:}\tau_2, x{:}\tau_1, \Gamma_2 \mathbin{@} \mathbf{r} \mathbin{\#} \langle t, s \rangle \mathbin{\#} \mathbf{q} \vdash e : \tau]\!] \;=\; \lambda(\mathbf{v_1}, y, x, \mathbf{v_2}).\ f\ (\mathbf{v_1}, x, y, \mathbf{v_2})} \quad (\textit{exch})$$

$$\frac{[\![\Gamma_1, y{:}\tau_1, z{:}\tau_1, \Gamma_2 \mathbin{@} \mathbf{r} \mathbin{\#} \langle s, t \rangle \mathbin{\#} \mathbf{q} \vdash e : \tau]\!] \;=\; f}{[\![\Gamma_1, x{:}\tau_1, \Gamma_2 \mathbin{@} \mathbf{r} \mathbin{\#} \langle s \oplus t \rangle \mathbin{\#} \mathbf{q} \vdash e[z, y \leftarrow x] : \tau]\!] \;=\; \lambda(\mathbf{v_1}, x, \mathbf{v_2}).\ f\ (\mathbf{v_1}, x|_s, x|_t, \mathbf{v_2})} \quad (\textit{contr})$$

Figure 19: Semantics of structural liveness

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 two ways of understanding coeffects are discussed further in Section 4.2.3.

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

a.) Ordinary, syntax-driven rules along with subcoeffecting

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

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

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

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

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

b.) Structural rules for context manipulation

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

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

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

Figure 20: Structural coeffect bounded reuse analysis

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. This has been captured in *bounded linear logic* by Girard et al. [40] who 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 6, the system preserves type and coeffect annotations under 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 and the annotations form a semiring with integer multiplication ($*$) for sequential composition and addition ($+$) for point-wise composition.

Variable access (*var*) annotates a variable with 1, meaning that it has been used once. An unused variable (*weak*) is annotated with 0. Multiple occurrences 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 point-wise 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 subcoeffecting rule (*sub*) safely overapproximates the number of accesses 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:

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

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:

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

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. [40] 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:

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

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

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 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 vector names $\mathbf{v}_i$ for contexts containing multiple variables i. e. have a type $\tau_1^{r_1} \times \ldots \times \tau_m^{r_m}$. Each vector contains multiple copies of each variable, to model the fact that variables are used in an affine way (at most once). 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 $s = \langle s_1, \ldots, s_n \rangle$. First, consider the (*contr*) rule:

$$\frac{\begin{array}{l}[\![\Gamma_1, y{:}\tau_1, z{:}\tau_1, \Gamma_2 \\ \quad @\,\mathbf{r} \# \langle s, t \rangle \# \mathbf{q} \vdash e : \tau]\!]\end{array} = \mathsf{f}}{\begin{array}{l}[\![\Gamma_1, x{:}\tau_1, \Gamma_2 @\,\mathbf{r} \# \langle s + t \rangle \# \mathbf{q} \\ \quad \vdash e[z, y \leftarrow x] : \tau]\!]\end{array} = \begin{array}{l}\lambda(\mathbf{v_1}, (x_1, \ldots, x_{s+t}), \mathbf{v_2}). \\ \quad \mathsf{f}\ (\mathbf{v_1}, (x_1, \ldots, x_s), (x_{s+1}, \ldots, x_{s+t}), \mathbf{v_2})\end{array}}$$

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:

$$\frac{\begin{array}{rcl}[\![\Gamma_1 @\,\mathbf{r} \vdash e_1 : \tau_1 \xrightarrow{t} \tau_2]\!] & = & \mathsf{f}_1 \\ [\![\Gamma_2 @\,\mathbf{s} \vdash e_2 : \tau_1]\!] & = & \mathsf{f}_2 \\ [\![\Gamma_1, \Gamma_2 @\,\mathbf{r} \times (t * \mathbf{s}) & & \\ \quad \vdash e_1\ e_2 : \tau_2]\!] & = & \end{array}}{\begin{array}{l}\lambda(\mathbf{v_1}, ((x_{1,1}, \ldots, x_{1,t*s_1}), \ldots, (x_{n,1}, \ldots, x_{n,t*s_n})). \\ \quad (\mathsf{f}_1\ \mathbf{v}_1) \\ \quad\quad (\ \mathsf{f}_2\ ((x_{1,1}, \ldots, x_{1,s_1}), \ldots, \\ \quad\quad\quad (x_{n,1}, \ldots, x_{1,s_n})\ ), \ldots, \\ \quad\quad\ \mathsf{f}_2\ ((x_{1,(t-1)*s_1+1}, \ldots, x_{1,t*s_1}), \ \ldots\ , \\ \quad\quad\quad (x_{n,(t-1)*s_n+1}, \ldots, x_{1,t*s_n})\ )\ )\end{array}}$$

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 is based on the idea of bounded linear logic (BLL) [40], but it is adapted to follow the structure of other coeffect systems discussed in this chapter. This elucidates the connection between BLL and coeffects.

The big difference, using the terminology from Section 2.3.3, is that our system is written in *language semantics* style, while BLL is written in *meta-language* style. We briefly consider the original BLL formulation.

The terms and types of our system are the terms and types of an 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 annotations to the variable context as a whole. The requirements are attached to individual variables and so our context $\tau_1, ..., \tau_n \mathbin{@} \langle k_1, ..., k_n \rangle$ corresponds to a BLL assumption $!_{k_1} A_1, ..., !_{k_n} A_n$. Using the formulation of bounded logic (and omitting the terms), the weakening and contraction rules are written as follows:

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

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* in Section 8.2.

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 ($+\!\!+$, $\langle - \rangle$) 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 6.

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. One benefit of our approach is that it lets us unify flat and structural systems (Section 8.1).

3.3.3 *Dataflow languages revisited*

When discussing dataflow languages in an earlier 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 an upper bound on past values for all variables. However, dataflow can also be adapted to a structural system which keeps the number of required past values individually for each variable. Consider the following example:

```
let offsetAdd = left + prev right
```

The value offsetAdd 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. A 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 dataflow system to a structural dataflow system is a change similar to the one between flat and structural liveness. In case of liveness analysis, we included the flat system only as an illustration (it is not practically useful). For dataflow, the flat system is less precise, but still practically useful (simplicity may outweigh precision).

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

$$(prev)\ \frac{\Gamma \mathbin{@} \mathbf{r} \vdash e : \tau}{\Gamma \mathbin{@} 1 + \mathbf{r} \vdash \mathtt{prev}\ e : \tau}$$

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

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

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

Figure 21: Structural coeffect bounded reuse analysis

TYPE SYSTEM. The type system in Figure 21 annotates the variable context with a vector of integers. This is similar to 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 dataflow, 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 dataflow, but different from bounded reuse and liveness (where unused variables have a different coeffect). Primitive requirements are introduced by the (*prev*) rule, which increments the annotations of all variables.

In flat dataflow, we identified sequential composition and pointwise composition as two primitive operations that were used in the (flat) application. In the structural system, these are used in (*app*) and (*contr*). 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 $\lambda x.\mathtt{prev}\ (y + x)$ applied to an argument $\mathtt{prev}\ (\mathtt{prev}\ y)$. 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:

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

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 $\mathtt{prev}\ (\mathtt{prev}\ y)$ 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:

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

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

`prev` $(y + (\texttt{prev}\ (\texttt{prev}\ y)))$. Semantically, this performs stream lookups $y[1]$ and $y[3]$ where the indices are the number of enclosing `prev` constructs.

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

The result is similar to 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:

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

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 consider first the semantics of the `prev` expression:

$$\frac{[\![\Gamma \mathbin{@} \langle s_1, \ldots, s_n \rangle \vdash e : \tau]\!] \ = \ f}{\begin{array}{l}[\![\Gamma \mathbin{@} \langle (s_1{+}1), \ldots, (s_n{+}1) \rangle \\ \quad \vdash \texttt{prev}\ e : \tau]\!]\end{array} \ = \ \begin{array}{l}\lambda((x_{1,0}, \ldots, x_{1,s_1+1}), \ldots, (x_{n,0}, \ldots, x_{n,s_n+1})). \\ \quad f\ ((x_{1,0}, \ldots, x_{1,s_1}), \ldots, (x_{n,0}, \ldots, x_{n,s_n}))\end{array}}$$

Here, 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 drops the last value. The following shows the semantics of contraction:

$$\frac{\begin{array}{l}[\![\Gamma_1, y : \tau_1, z : \tau_1, \Gamma_2 \\ \quad @\, \mathbf{r} \# \langle s, t \rangle \# \mathbf{q} \vdash e : \tau]\!]\end{array} \ = \ f}{\begin{array}{l}[\![\Gamma_1, x : \tau_1, \Gamma_2 \mathbin{@} \mathbf{r} \# \langle \max(s,t) \rangle \# \mathbf{q} \\ \quad \vdash e[z, y \leftarrow x] : \tau]\!]\end{array} \ = \ \begin{array}{l}\lambda(\mathbf{v_1}, (x_0, x_1, \ldots, x_{\max(s,t)}), \mathbf{v_2}). \\ \quad f\ ((\mathbf{v_1}, (x_0, \ldots, x_s), (x_0, \ldots, x_t), \mathbf{v_2}))\end{array}}$$

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 dataflow) 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 statically scoped properties, i. e. related to variables.

The dataflow 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 13).

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 using a runtime mark (e. g. in the Perl language) or using a static type system. Tainting can be viewed as a special case of *provenance tracking*, known from database systems [20], where values are annotated with more detailed information about their source.

Type systems based on tainting have been use to prevent cross-site scripting attacks [117] and SQL injection attacks [44, 43]. In the latter case, we want to check that SQL commands cannot be directly constructed from unchecked 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 an 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 with a Boolean flag (for tainting) or more complex data (for provenance). In static checking, the information needs to be associated with the variables in the variable context.

CORE DEPENDENCY CALCULUS. Taint checking 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 a credit card number) cannot be leaked to contexts with low secrecy (e. g. sent via an unsecured network channel). Volpano et al. [118] provide the first (provably) sound type system that guarantees non-inference and Sabelfeld et al. [93] survey more recent work. Information flow checking has been also integrated (as a single-purpose extension) in the FlowCaml [99] language. Finally, Russo et al. and Swamy et al. [92, 101] show that such 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 partially ordered finite set of classes. For example a set $\{L, H\}$ represents low and high secrecy, respectively with $L \leqslant H$ meaning that low security values can be treated as high security (but not the other way round).

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:

```
let z = if x > 0 then y 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).

Although we do not describe a coeffect calculus for information flow checking, it is worth noting that Abadi et al. [1] realized that there is a number of analyses similar to secure information flow and unified them using a single model called Dependency Core Calculus (DCC). This would be a useful basis for coeffect-based information flow checking.

The DCC captures other cases where some information about expression relies on properties of variables in the context where it executes. This includes, for example, *binding time analysis* [109], which detects which parts of programs can be partially evaluated (do not depend on user input) and *program slicing* [110] 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 taint tracking, security and provenance in detail, 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*. Accessing a variable marks it as untainted, but using an expression that depends on some variable in certain dangerous contexts – such as in arguments of query – does introduce a taint on all the variables contributing to the expression. This is captured using the standard application rule (*app*):

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

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.

A number of systems capture contextual properties, but make it possible to *change* the context – not just by evaluating certain code block in a locally modified context (e. g. by wrapping it in prev in dataflow), but also by calling a function that acquires new capabilities and returns those to the caller. Such actions appear to be closer to effects than to coeffects. 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. [27] 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 [111] who developed an effect system (as discussed in Section 2.3.2) that uses lexically scoped *memory regions* to provide an efficient and controlled memory management.

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

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

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

```
let calculate = λinput →
  letrgn ρ in
  let x = ref_ρ 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 from systems with passive context. To type check such programs, we not only need to know what context is required to call `calculate` (i. e. context on the left-hand side of ⊢). We also need to know what effects an expression has on the context when it evaluates and the current context needs to be updated after a function call. This is an effectful property that would appear on the right-hand side of ⊢.

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 (with a comonad to model coeffects on the left):

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

Systems with active contexts require two annotations – one specifies the context required before the call and one specifies how the context changes after the call (this could be a *new* context or an *update* to the original context). Thus the structure of the semantics would look as follows (with a comonad to model coeffects on the left and a monad to model effects on the right):

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

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, capturing active contexts is an interesting future work.

SOFTWARE UPDATING. Another example of a system that uses contextual information actively is dynamic software updating (DSU) [35, 47]. DSU systems have the ability to update programs at runtime without stopping them. For example, Proteus developed by Stoyle et al. [100] investigates what language support is needed to enable safe dynamic software updating in C-like languages. The work is based on capabilities and follows a structure similar to the Calculus of Capabilities [27].

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 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 principled approaches based on comonads and modal S4, ad-hoc type system extensions and static analyses as well as approaches based on monads. We looked at a number of applications including Haskell's implicit parameters and type classes, dataflow 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 just by instantiating our unified calculi.

This chapter was divided into two sections. First, we looked at *flat* systems, which track whole-context properties. Next, we looked at *structural* systems, which track per-variable properties. Both are useful and important – for example, implicit parameters can only be expressed as a *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 6). We can, however, unify the two variants into a single system discussed in Section 8.1.

Part II

COEFFECT CALCULI

In this part, we capture the similarities between the concrete context-aware languages presented in the previous chapter. We also develop the key novel technical contributions of the thesis. We define a *flat coeffect type system* (Chapter 4) that is parameterized by a *coeffect algebra* and a mechanism for choosing unique typing derivation. We instantiate a coeffect type system with a concrete coeffect algebra and procedure for choosing unique typing derivation for three languages to capture dataflow, implicit parameters and liveness.

The type system is complemented with a translational semantics for coeffect-based context-aware programming languages (Chapter 5). The semantics is inspired by a categorical model based on *indexed comonads* and it translates a source context-aware program into a target program in a simple functional language with comonadically-inspired primitives. We give a concrete definition of the primitives for dataflow, implicit parameters and liveness and present a syntactic safety proof for these three languages.

The following page provides a detailed overview of the content of Chapters 4 and Chapters 5, highlighting the split between general definitions and properties (about the coeffect calculus) and concrete definitions and properties (about concrete context-awre language). Chapter 6 mirrors the same development for *structural coeffect systems*.

Chapter 4		
	Coeffect calculus	Language-specific
Syntax	Coeffect λ-calculus (Section 4.2)	Extensions such as `?param` and `prev` (Section 4.2.4)
Type system	Abstract coeffect algebra (Section 4.2.1)	Concrete instances of the coeffect algebra (Section 4.2.4)
	Coeffect type system parameterized by the coeffect algebra (Section 4.2.2)	Typing for language-specific extensions (Section 4.2.4)
		Procedure for determining a unique typing derivation (Section 4.3)
Properties	Syntactic properties of coeffect λ-calculus (Section 4.4)	Uniqueness of the above (Section 4.3)

Chapter 5		
	Coeffect calculus	Language-specific
Categorical	Indexed comonads (Section 5.2.4)	Examples including indexed product, list and maybe comonads (Section 5.2.5)
	Categorical semantics of coeffect λ-calculus (Section 5.2.6)	
Translational	Functional target language (Section 5.3.1)	
	Translation from coeffect λ-calculus to target language (Section 5.3.3)	Translation for language-specific extensions (`prev`, `?p`) (Sections 5.4.1 and 5.4.2)
Operational	Abstract comonadically-inspired primitives (Section 5.3.3)	Concrete reduction rules for comonadically-inspired primitives (Sections 5.4.1 and 5.4.2)
		Reduction rules for language-specific extensions (`prev`, `?p`) (Sections 5.4.1 and 5.4.2)
	Sketch of generalized syntactic soundness (Section 5.5)	Syntactic soundness (Sections 5.4.1 and 5.4.2)

4 TYPES FOR FLAT COEFFECTS

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 systems* capturing whole-context properties and *structural systems* capturing per-variable properties. As we show in Section 8.1, the systems can be further unified using a single abstraction, 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, we discuss *flat coeffects* (Chapter 4 and Chapter 5) and *structural coeffects* (Chapter 6) separately.

In this chapter, we develop a *flat coeffect calculus* that provides a type system for tracking per-context properties of context-aware programming languages. The *coeffect calculus* captures the shared properties of such languages. It is parameterized by a *flat coeffect algebra* and can be instantiated to track implicit parameters, liveness and number of required past values in dataflow languages. To capture contextual properties in full generality, the flat coeffect calculus permits multiple valid typing derivations for a given term. To resolve the ambiguity arising from such generality, each concrete context-aware language is also equipped with an algorithm for choosing a unique typing derivation. This allows us to explore the language design landscape, while still following the usual scoping rules for languages with established approaches (e.g. implicit parameters in GHC).

In the next chapter, we give operational meaning for concrete coeffect languages based on the flat coeffect calculus and we discuss their safety.

CHAPTER STRUCTURE AND CONTRIBUTIONS

- We present a *flat coeffect calculus* as a type system that is parameterized by a *flat coeffect algebra* (Section 4.2). We show that the system can be instantiated to obtain three of the systems discussed in Section 3.2, namely implicit parameters, liveness and dataflow.
- The coeffect calculus permits multiple typing derivations due to the ambiguity inherent in contextual lambda abstraction. Each concrete context-aware language based on the coeffect calculus must specify how such ambiguities are to be resolved. We give the procedures for choosing unique typing derivations for our three examples (Section 4.3).
- We discuss equational properties of the calculus, covering 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).

4.1 INTRODUCTION

In the previous chapter, we looked at three examples of systems that track whole-context properties. The type systems for whole-context liveness (Section 3.2.3) and whole-context dataflow (Section 3.2.4) have a similar structure in two ways. First, lambda abstraction duplicates their *context demands*. Given a body with context demands r, the declaration site context *as well*

as the function arrow are annotated with r. Second, the context demands in the type systems are combined using two different operators (representing sequential and pointwise operations).

The system for tracking implicit parameters (Section 3.2.1) differs. In lambda abstraction, it partitions the context demands between the declaration site and the call site. Furthermore, the operator that combines context demands is $\cup$ for both sequential and pointwise composition.

Despite the differences, the systems fit the same framework. This becomes apparent when we consider the categorical structure (Chapter 5). Rather than starting from the categorical semantics, we first explain how the systems can be unified syntactically (Section 4.1.1) and then provide the semantics as an additional justification.

The development in this chapter can be seen as a counterpart to the well-known development of *effect systems* [38]. Chapter 5 then links *coeffects* with *comonads* in the same way in which effect systems have been linked with monads [67]. The syntax and type system of the flat coeffect calculus follows a similar style to that of effect systems [62, 106], but differs in the structure of lambda abstraction as discussed briefly here and in Section 3.1.1 (the relationship with monads is further discussed in Section 5.6).

4.1.1 *A unified treatment of lambda abstraction*

Recall the lambda abstraction rules for the implicit parameters coeffect system (annotating contexts with sets of required parameters) and the dataflow system (annotating contexts with the number of past required values):

$$(\textit{param})\ \frac{\Gamma, x{:}\tau_1 \mathbin{@} r \cup s \vdash e : \tau_2}{\Gamma \mathbin{@} r \vdash \lambda x.e : \tau_1 \xrightarrow{s} \tau_2} \qquad (\textit{df1})\ \frac{\Gamma, x{:}\tau_1 \mathbin{@} n \vdash e : \tau_2}{\Gamma \mathbin{@} n \vdash \lambda x.e : \tau_1 \xrightarrow{n} \tau_2}$$

In order to capture both systems using a single calculus, we need a way of rewriting the (*df1*) rule such that the annotation in the assumption is in the form $n \circ m$ for some operation $\circ$. For the dataflow system, this can be achieved by using the *min* function:

$$(\textit{df2})\ \frac{\Gamma, x{:}\tau_1 \mathbin{@} \min(n, m) \vdash e : \tau_2}{\Gamma \mathbin{@} n \vdash \lambda x.e : \tau_1 \xrightarrow{m} \tau_2}$$

The rule (*df1*) is admissible in a system that includes the (*df2*) rule. That is, a typing derivation using (*df1*) is also valid when using (*df2*). Furthermore, if we include sub-typing rule (on annotations of functions) and subcoeffecting rule (on annotations of contexts), then the reverse is also true – because $\mathit{min}(n, m) \leqslant m$ and $\mathit{min}(n, m) \leqslant n$. In other words (*df2*) permits an implicit subcoeffecting (and sub-typing) that is not possible when using the (*df1*) rule, but it has a structure that can be unified with (*param*).

4.2 FLAT COEFFECT CALCULUS

This section describes the *flat coeffect calculus*. A small programming language based on the λ-calculus with a type system that statically tracks context demands. The calculus can capture different notions of context. The structure of context demands is provided by a *flat coeffect algebra* (defined in the next section) which is an abstract algebraic structure that can be instantiated to model concrete context demands (sets of implicit parameters, number of past values as integers or other information). Annotations that specify context demands are written as r, s, t.

We enrich types and typing judgements with coeffect annotations r, s, t; typing judgements are written as $\Gamma \mathbin{@} r \vdash e : \tau$. The expressions of the calculus are those of the λ-calculus with *let* binding. We also include a type num as an example of a concrete base type with numerical constants written as n:

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

Note that the lambda abstraction in the syntax is written in the Church style and requires a type annotation. This will be used in Section 4.3 where we discuss how to find a unique typing derivation for context-aware computations. Using Church style lambda abstraction, we can directly focus on the more interesting problem of finding unique *coeffect annotations* rather than solving the problem of type reconstruction.

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 categorically-inspired translation in the next chapter, the function $\tau_1 \xrightarrow{r} \tau_2$ is translated into a function $C^r \tau_1 \to \tau_2$. However, the type constructor C^r does not itself exist as a syntactic value in the coeffect calculus. 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 has been discussed in Section 2.3.1). The annotations r are formed by an algebraic structure discussed next.

4.2.1 *Flat coeffect algebra*

To make the flat coeffect system general enough, the algebra consists of three operations. Two of them, $\circledast$ and $\oplus$, represent *sequential* and *pointwise* composition, which are mainly used in function application. The third operator, $\wedge$ is used in lambda abstraction and represents *splitting* of context demands.

In addition to the three operations, the algebra also requires 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* $\mathsf{use}, \mathsf{ign} \in \mathcal{C}$*, binary relation* $\leqslant$ *and binary operations* $\circledast, \oplus, \wedge$ *such that* $(\mathcal{C}, \circledast, \mathsf{use})$ *is a monoid,* $(\mathcal{C}, \oplus, \mathsf{ign})$ *is an idempotent monoid,* $(\mathcal{C}, \wedge)$ *is a band (idempotent semigroup) and* $(\mathcal{C}, \leqslant)$ *is a pre-order. That is, for all* $r, s, t \in \mathcal{C}$*:*

$$\begin{array}{lll} r \circledast (s \circledast t) = (r \circledast s) \circledast t & & \mathsf{use} \circledast r = r = r \circledast \mathsf{use} \\ r \oplus (s \oplus t) = (r \oplus s) \oplus t & r \oplus r = r & \mathsf{ign} \oplus r = r = r \oplus \mathsf{ign} \\ r \wedge (s \wedge t) = (r \wedge s) \wedge t & r \wedge r = r & \\ \text{if } r \leqslant s \text{ and } s \leqslant t \text{ then } r \leqslant t & t \leqslant t & \end{array}$$

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 in the dataflow system all three are distinct. Similarly, the two special elements coincide in some, but not all systems. The required axioms 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 monoid axioms are required in order to form a category structure in the semantics (Section 5.2).
- The idempotent monoid $(\mathcal{C}, \oplus, \mathsf{ign})$ represents *pointwise* composition, i.e. the case when the same context is passed to multiple (independent) computations. The monoid axioms guarantee that usual syntactic transformations on tuples and the unit value (Section 4.5) preserve the coeffect. Idempotence holds for all our examples and allows us to unify the flat and structural systems in Section 8.1.
- For the $\wedge$ operation, we require associativity and idempotence. The idempotence demand makes it possible to duplicate the given coeffects and place the same demand on both call site and declaration site. Using the example from Section 4.1.1, this guarantees that the rule (*df1*) is not a special case, but can always be derived from (*df2*). In some cases, the operator forms a monoid with the unit being the greatest element of the set $\mathcal{C}$.

It is worth noting that, in some of the systems, the operators $\oplus$ and $\wedge$ are the least upper bound and the greatest lower bounds of a lattice. For example, in dataflow computations, they are *max* and *min* respectively. However, this duality does not hold for implicit parameters.

SEMIRING. The $\circledast$ and $\oplus$ operations of the flat coeffect algebra together with the use and ign values resemble semiring. As required by semiring laws, both of the operations are associative; use and ign are the units of $\circledast$ and $\oplus$, respectively, and the distributivity laws also hold. The two additional laws of semiring that are not required by a flat coeffect algebra are symmetry of $\circledast$ and annihilation:

$$\mathsf{ign} \circledast r = \mathsf{ign} = r \circledast \mathsf{ign} \qquad r \oplus s = s \oplus r$$

The symmetry of $\circledast$ holds for all our examples, but we do not include it so that we do not unnecessarily restrict the system. The annihilation holds for some of the examples (liveness and optimized dataflow in Section 4.2.4), but not for all of them. In particular, it does not hold for implicit parameters $\emptyset \cup r \neq \emptyset$ and so requiring it would lead to a more restrictive structure than is desirable.

ORDERING. The flat coeffect algebra includes a pre-order relation $\leqslant$. This will be used to introduce subcoeffecting and subtyping in Section 4.5.1, but we make it a part of the flat coeffect algebra, as it will be useful for characterization of different kinds of coeffect calculi. When the idempotent monoid $(\mathcal{C}, \oplus, \mathsf{ign})$ is also commutative (i.e. forms a semi-lattice), the $\leqslant$ relation can be defined as the ordering of the semi-lattice:

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

This definition is consistent with all three examples that motivate flat coeffect calculus, but it cannot be used with the structural coeffects (Chapter 6), where it fails for the bounded reuse calculus. For this reason, we choose not to use it for flat coeffect calculus either.

Furthermore, the use coeffect is often the top or the bottom element of the semi-lattice. As discussed in Section 4.4, when this is the case, we are able to prove certain syntactic properties of the calculus.

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

$$(const)\ \frac{}{\Gamma\,@\,\mathsf{ign} \vdash n : \mathsf{num}}$$

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

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

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

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

Figure 22: Type system for the flat coeffect calculus

4.2.2 *Type system*

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.

The (*abs*) rule is defined as discussed in Section 4.1.1. The body is annotated with context demands $r \wedge s$, which are then split between the context-demands on the declaration site r and context-demands on the call site s.

In function application (*app*), context demands of both expressions and the function are combined. As discussed in Chapter 3, sequential composition is used to combine the context-demands of the argument s with the context-demands of the function t. The result $s \circledast t$ is then composed using pointwise composition with the context demands of the expression that represents the function r, giving the coeffect $r \oplus (s \circledast t)$.

The type system also includes a rule for let-binding. The rule is *not* equivalent to the derived rule for $(\lambda x.e_2)\ e_1$, but it corresponds to *one* possible typing derivation. As we show in 4.5.2, the typing used in (*let*) is more precise than the general rule that can be derived from $(\lambda x.e_2)\ e_1$.

To guide understanding of the system, we also show a non-syntax-directed (*sub*) rule for subcoeffecting. The rule states that an expression with context demands r' can be treated as an expression with greater context demands r. We return to subcoeffecting, subtyping and additional constructs such as pairs in Section 4.5. When discussing the procedure for choosing unique typing in Section 4.3, we consider only the syntax-directed part of the system.

4.2.3 *Understanding flat coeffects*

Before proceeding, let us clarify how the typing judgements should be understood. The coeffect calculus can be understood in two ways discussed in this and the next chapter. As a type system (Chapter 4), it provides analysis of context dependence. As a semantics (Chapter 5), it specifies how context is propagated. These two readings provide different ways of interpreting the judgement $\Gamma\,@\,r \vdash e : \tau$ and the typing rules used to define it.

- ANALYSIS OF CONTEXT DEPENDENCE. Syntactically, coeffect annotations r model *context demands*. This means we can over-approximate them and require more in the type system than is needed at runtime.

  Syntactically, the typing rules are best read top-down (from assumptions to the consequent). In function application, the context demands of multiple assumptions (arising from two sub-expressions) are *merged*; in lambda abstraction, the demands of a single expression (the body) 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 (from the consequent to assumptions). 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.

For example, using the syntactic reading, the operators $\wedge$ and $\oplus$ represent *merging* and *splitting* of context demands – in the (*abs*) rule, $\wedge$ appears in the assumption and the combined context demands of the body are split between two positions in the conclusions; in the (*app*) rule, $\oplus$ appears in the conclusion and merges two context demands from the assumptions.

The reason for this asymmetry follows from the fact that the context appears in a *negative position* in the semantic model (Section 5.2). It means that we need to be careful about using the words *split* and *merge*, because they can be read as meaning exactly the opposite things. To disambiguate, we always use the term *context demands* when using the syntactic view, especially in the rest of Chapter 4, and *context capabilities* or just *available context* when using the semantic view, especially in Chapter 5.

4.2.4 *Examples of flat coeffects*

The flat coeffect calculus generalizes the three 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.

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

For simplicity, assume that all parameters have the same type num and so the annotations only track sets of names. The definition uses a set union for all three operations. Both variables and constants are annotated with $\emptyset$ and the ordering is defined by $\subseteq$. The definition satisfies the flat coeffect algebra axioms because $(S, \cup, \emptyset)$ is an idempotent, commutative monoid. The language has additional syntax for defining an implicit parameter and for accessing it, together with associated typing rules:

$$e ::= \ldots \mid ?\mathsf{p} \mid \mathtt{let}\ ?\mathsf{p}\ =\ e_1\ \mathtt{in}\ e_2$$

$$(\textit{param})\ \frac{}{\Gamma\,@\,\{?\mathsf{p}\} \vdash ?\mathsf{p} : \mathsf{num}}$$

$$(\textit{letpar})\ \frac{\Gamma\,@\,r \vdash e_1 : \tau_1 \qquad \Gamma\,@\,s \vdash e_2 : \tau_2}{\Gamma\,@\,r \cup (s \setminus \{?\mathsf{p}\}) \vdash \mathtt{let}\ ?\mathsf{p} = e_1\ \mathtt{in}\ e_2 : \tau_2}$$

The (*param*) rule specifies that the accessed parameter ?p needs to be in the set of required parameters r. As discussed earlier, we use the same type num for all parameters, but it is also possible to define a coeffect calculus that uses mappings from names to types.

The (*letpar*) rule is the same as the one discussed in Section 3.2.1. As both of the rules are specific to implicit parameters, we write the operations on coeffects directly using set operations – coeffect-specific operations such as set subtraction are not a part of the unified coeffect algebra.

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

The liveness example is interesting because it does not require any additional syntactic extensions to the language. It annotates constants and variables with D and L, respectively and it captures how those annotation propagate through the remaining language constructs.

As in Section 3.2.3, sequential composition $\circledast$ is modelled by the meet operation $\sqcap$ and pointwise composition $\oplus$ is modelled by join $\sqcup$. The two-point lattice induces a commutative, idempotent monoid. Distributivity $(r \sqcup s) \sqcap t = (r \sqcap t) \sqcup (s \sqcap t)$ does not hold for *every* lattice, but it trivially holds for the 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. The latter is less precise, but it is a valid derivation that could also be obtained via sub-typing.

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

As discussed earlier, sequential composition $\circledast$ is represented by $+$ and pointwise composition $\oplus$ uses *max*. For dataflow, we need a third separate operator for lambda abstraction. Annotating the body with $min(r, s)$ ensures that both call site and declaration site annotations are equal or greater than the annotation of the body.

As required by the axioms, $(\mathbb{N}, +, 0)$ and $(\mathbb{N}, max, 0)$ form monoids and $(\mathbb{N}, min)$ forms a band. Note that dataflow is our first example where $\circledast$ is not idempotent. The distributivity axioms require addition to distribute over maximum, which is easy to see.

A simple dataflow language includes an additional construct `prev` for accessing the previous value in a stream with an additional typing rule that looks as follows:

$$e ::= \ldots \mid \texttt{next}\ e$$

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

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

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

Note that $(\mathbb{N} \cup \{-\infty\}, +, 0)$ is a monoid for the extended definition of $+$; for the bottom element $0 + \bot = \bot$ and for natural numbers $0 + n = n$. The structure $(\mathbb{N}, max, \bot)$ is also a monoid, because $\bot$ is the least element and so $max(n, \bot) = n$. Finally, $(\mathbb{N}, min)$ is a band and the distributivity axioms also hold for $\mathbb{N} \cup \{-\infty\}$.

4.3 CHOOSING A UNIQUE TYPING

As discussed in Chapter 3, the lambda abstraction rule for coeffect systems differs from the rule for effect systems in that it does not delay all context demands. In case of implicit parameters (Section 3.2.1), the demands can be satisfied either by the call site or by the declaration site. In case of dataflow and liveness, the rule discussed in Section 4.2 reintroduces similar ambiguity because it allows multiple valid typing derivations.

Furthermore, the semantics of context-aware languages in Chapter 3 and also in Chapter 5 is defined over *typing derivations* and so the meaning of a program depends on the typing derivation chosen. In this section, we specify how to choose the desired *unique* typing derivation in each of the coeffect systems we consider.

The most interesting case is that of implicit parameters. Consider the following program written using the extended coeffect calculus:

```
let ?x = 1
let f  = λy. ?x in
let ?x = 2 in f 0
```

There are two possible typings allowed by the typing rules discussed in Section 4.2.2 that lead to two possible meanings of the program – evaluating to 1 and 2, respectively:

- $\mathsf{f} : \mathsf{num} \xrightarrow{\emptyset} \mathsf{num}$ – in this case, the value of ?x is captured from the declaration site and the program produces 1.
- $\mathsf{f} : \mathsf{num} \xrightarrow{\{?x\}} \mathsf{num}$ – in this case, the parameter ?x is required from the call site and the program produces 2.

The coeffect calculus intentionally allows both options, acknowledging the fact that the choice needs to be made for each individual concrete context-aware programming language. In the above case, one typing derivation represents dynamic binding and the other static binding, but more subtleties arise when the nested expression uses multiple implicit parameters.

In this section, we discuss the specific choices of typing derivation for implicit parameters, dataflow and liveness. We use the fact that the coeffect calculus uses Church style syntax for lambda abstraction giving a type annotation for the bound variable. This does not affect the handling of coeffects (those are not defined by the type annotation), but it lets us prove *uniqueness of typing*; a theorem showing that we define a *unique* way of assigning coeffects to otherwise well-typed programs.

4.3.1 *Implicit parameters*

For implicit parameters we choose to follow the behaviour implemented by GHC [60] where function abstraction captures all parameters that are statically available at the declaration site and places all other demands on the call site. For the example above, this means that the body of f captures

$$(var)\ \frac{x : \tau \in \Gamma}{\Gamma; \Delta \mathbin{@} \mathsf{use} \vdash x : \tau}$$

$$(const)\ \frac{}{\Gamma; \Delta \mathbin{@} \mathsf{ign} \vdash n : \mathsf{num}}$$

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

$$(let)\ \frac{\Gamma; \Delta \mathbin{@} r \vdash e_1 : \tau_1 \qquad \Gamma, x{:}\tau_1; \Delta \mathbin{@} s \vdash e_2 : \tau_2}{\Gamma; \Delta \mathbin{@} s \oplus (s \circledast r) \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau_2}$$

$$(param)\ \frac{}{\Gamma; \Delta \mathbin{@} \{?p\} \vdash ?p : \mathsf{num}}$$

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

$$(letpar)\ \frac{\Gamma; \Delta \mathbin{@} r \vdash e_1 : \mathsf{num} \qquad \Gamma; \Delta \cup \{?p\} \mathbin{@} s \vdash e_2 : \tau}{\Gamma; \Delta \mathbin{@} r \cup (s \setminus \{?p\}) \vdash \mathtt{let}\ ?p = e_1\ \mathtt{in}\ e_2 : \tau}$$

Figure 23: Choosing unique typing for implicit parameters

the value of ?p available from the declaration site and f will be typed as a function requiring no parameters (coeffect $\emptyset$). The program thus evaluates to a numerical value 1.

To express this behaviour formally, we extend the coeffect type system to additionally track implicit parameters that are currently in static scope. The typing judgement becomes:

$$\Gamma; \Delta \mathbin{@} r \vdash e : \tau$$

Here, Δ is a set of implicit parameters that are in scope at the declaration site. The modified typing rules are shown in Figure 23. The rules (*var*), (*const*), (*app*) and (*let*) are modified to use the new typing judgement, but they simply propagate the information tracked by Δ to all assumptions. The (*param*) rule also remains unchanged – the implicit parameter access is still tracked by the coeffect r meaning that we still allow a form of dynamic binding (the parameter does not have to be in static scope).

The most interesting rule is (*abs*). The body of a function requires implicit parameters tracked by r and the parameters currently in (static) scope are Δ. The coeffect on the declaration site becomes Δ (capture all available parameters) and the latent coeffect attached to the function becomes $r \setminus \Delta$ (require any remaining parameters from the call site). Finally, in the (*letpar*) rule, we add the newly bound implicit parameter ?p to the static scope in the sub-expression e_2.

PROPERTIES. If a program written in a coeffect language with implicit parameters is well-typed in the type system presented in Figure 23 then this identifies the unique preferred derivation for the program. We use this unique typing derivation to give the semantics of a coeffect language with implicit parameters in Chapter 5 and we also implement this algorithm as discussed in Chapter 7.

The type system is more restrictive than the fully general one and it rejects certain programs that could be typed using the more general system. This

is expected – we are restricting the fully general coeffect calculus to match the typing and semantics of implicit parameters as known from Haskell.

In order to prove the uniqueness of typing theorem (Theorem 2), we follow the standard approach [88] and first give the inversion lemma (Lemma 1).

Lemma 1 (Inversion lemma for implicit parameters). *For the type system defined in Figure 23:*

1. *If* $\Gamma;\Delta@c \vdash x : \tau$ *then* $x : \tau \in \Gamma$ *and* $c = \emptyset$.
2. *If* $\Gamma;\Delta@c \vdash n : \tau$ *then* $\tau = \mathsf{num}$ *and* $c = \emptyset$.
3. *If* $\Gamma;\Delta@c \vdash e_1\ e_2 : \tau_2$ *then there is some* τ_1, r, s *and* t *such that* $\Gamma;\Delta@r \vdash e_1 : \tau_1 \xrightarrow{t} \tau_2$ *and* $\Gamma;\Delta@s \vdash e_2 : \tau_1$ *and also* $c = r \cup s \cup t$.
4. *If* $\Gamma;\Delta@c \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau_2$ *then there is some* τ_1, s *and* r *such that* $\Gamma;\Delta@r \vdash e_1 : \tau_1$ *and* $\Gamma, x{:}\tau_1;\Delta@s \vdash e_2 : \tau_2$ *and also* $c = s \cup r$.
5. *If* $\Gamma;\Delta@c \vdash ?p : \mathsf{num}$ *then* $?p \in c$ *and* $c = \{?p\}$.
6. *If* $\Gamma;\Delta@c \vdash \lambda x{:}\tau_1.e : \tau$ *then there is some* τ_2 *such that* $\tau = \tau_1 \xrightarrow{s} \tau_2$ $\Gamma, x{:}\tau_1;\Delta@r \vdash e : \tau_2$ *and* $c = \Delta$ *and also* $s = r \setminus \Delta$.
7. *If* $\Gamma;\Delta@c \vdash \mathtt{let}\ ?p = e_1\ \mathtt{in}\ e_2 : \tau$ *then there is some* r, s *such that* $\Gamma;\Delta@r \vdash e_1 : \mathsf{num}$ *and* $\Gamma;\Delta \cup \{?p\}@s \vdash e_2 : \tau$ *and also* $c = r \cup (s \setminus \{?p\})$.

Proof. Follows from the individual rules given in Figure 23. □

Theorem 2 (Uniqueness of coeffect typing for implicit parameters). *In the type system for implicit parameters defined in Figure 23, when* $\Gamma;\Delta@r \vdash e : \tau$ *and* $\Gamma;\Delta@r' \vdash e : \tau'$ *then* $\tau = \tau'$ *and* $r = r'$.

Proof. Suppose that (A) $\Gamma;\Delta@c \vdash e : \tau$ and (B) $\Gamma;\Delta@c' \vdash e : \tau'$. We show by induction over the typing derivation of $\Gamma;\Delta@c \vdash e : \tau$ that $\tau = \tau'$ and $c = c'$.

Case (*abs*): $e = \lambda x : \tau_1.e_1$ and $c = \Delta$. $\tau = \tau_1 \xrightarrow{r\setminus\Delta} \tau_2$ for some r, τ_2 and also $\Gamma, x : \tau_1;\Delta@r \vdash e : \tau_2$. By case (6) of Lemma 1, the final rule of the derivation (B) must have also been (*abs*) and this derivation has a sub-derivation with a conclusion $\Gamma, x{:}\tau_1;\Delta@r \vdash e : \tau_2'$. By the induction hypothesis $\tau_2 = \tau_2'$ and $c = c'$ and therefore $\tau = \tau'$.

Case (*param*): $e = ?p$, from Lemma 1, $\tau = \tau' = \mathsf{int}$ and $c = c' = \{?p\}$.

Cases (*var*), (*const*) are direct consequence of Lemma 1.

Cases (*app*), (*let*) and (*letpar*) similarly to (*abs*). □

Finally, we note that unique typing derivations obtained using the type system given in Figure 23 are valid typing derivations under the original flat coeffect type system in Figure 22.

Theorem 3 (Admisibility of unique typing for implicit parameters). *If* $\Gamma;\Delta@r \vdash e : \tau$ *(using the rules in Figure 23) then also* $\Gamma@r \vdash e : \tau$ *(using the rules in Figure 22 and Example 1).*

Proof. Each typing rule in the unique type derivation is a special case of the corresponding typing rule in the flat coeffect calculus (ignoring the additional context Δ); the splitting of coeffects in (*abs*) in Figure 23 is a special case of splitting two sets using $\cup$. □

4.3.2 *Dataflow and liveness*

Resolving the ambiguity for liveness and dataflow computations is easier than for implicit parameters. It suffices to use a lambda abstraction rule that duplicates the coeffects of the body:

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

This is the rule that we originally used for liveness and dataflow computations in Chapter 3. This rule cannot be used with implicit parameters and so the additional flexibility provided by the $\wedge$ operator is needed in the general flat coeffect calculus.

For liveness and dataflow, the (*idabs*) rule provides the most precise coeffect. Assume we have a lambda abstraction with a body that has coeffects r. The ordinary (*abs*) rule requires us to find s, t such that $r = s \wedge t$.

- For dataflow, this is $r = \mathit{min}(s, t)$. The smallest s, t such that the equality holds are $s = t = r$.
- For liveness, this is $r = s \sqcup t$. When $r = \mathsf{L}$, the only solution is $s = t = \mathsf{L}$; when $r = \mathsf{D}$, the most precise solution is $s = t = \mathsf{D}$ because $\mathsf{D} \sqsubseteq \mathsf{L}$.

The notion of "more precise" solution can be defined in terms of subcoeffecting and subtyping. We return to this topic in Section 4.5.3 and we also precisely characterise for which coeffect system is the (*idabs*) rule preferable over the (*abs*) rule.

PROPERTIES. If a program written in a coeffect language for liveness or dataflow is well-typed according to the type system presented in Figure 22 with the (*abs*) rule replaced by (*idabs*), then the type system gives a unique derivation. As for implicit parameters, this defines the semantics of coeffect program (Chapter 5) and it is used in the implementation (Chapter 7).

We note that the unique typing derivation is admissible in the original coeffect type system. For dataflow and liveness, this follows directly from the fact that (*idabs*) is a special case of the (*abs*) rule and so we do not state this explicitly as in Theorem 3 for implicit parameters.

In order to prove the uniqueness of typing theorem (Theorem 5), we first need the inversion lemma (Lemma 4).

Lemma 4 (Inversion lemma for liveness and dataflow)**.** *For the type system defined in Figure 22 with the (abs) rule replaced by (idabs):*

1. *If* $\Gamma \mathbin{@} c \vdash x : \tau$ *then* $x : \tau \in \Gamma$ *and* $c = \mathsf{use}$.
2. *If* $\Gamma \mathbin{@} c \vdash n : \tau$ *then* $\tau = \mathsf{num}$ *and* $c = \mathsf{ign}$.
3. *If* $\Gamma \mathbin{@} c \vdash e_1\ e_2 : \tau_2$ *then there is some* τ_1, r, s *and* t *such that* $\Gamma \mathbin{@} r \vdash e_1 : \tau_1 \xrightarrow{t} \tau_2$ *and* $\Gamma \mathbin{@} s \vdash e_2 : \tau_1$ *and also* $c = r \oplus (s \circledast t)$.
4. *If* $\Gamma \mathbin{@} c \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau_2$ *then there is some* τ_1, s *and* r *such that* $\Gamma \mathbin{@} r \vdash e_1 : \tau_1$ *and* $\Gamma, x{:}\tau_1 \mathbin{@} s \vdash e_2 : \tau_2$ *and also* $c = s \oplus (s \circledast r)$.
5. *If* $\Gamma \mathbin{@} c \vdash \lambda x{:}\tau_1.e : \tau$ *then there is some* τ_2 *such that* $\tau = \tau_1 \xrightarrow{c} \tau_2$ *and* $\Gamma, x{:}\tau_1 \mathbin{@} c \vdash e : \tau_2$.

Proof. Follows from the individual rules given in Figure 23. □

Theorem 5 (Uniqueness of coeffect typing for liveness and dataflow)**.** *In the type system for liveness and dataflow defined in Figure 22 with the (abs) rule replaced by (idabs), when* $\Gamma \mathbin{@} r \vdash e : \tau$ *and* $\Gamma \mathbin{@} r' \vdash e : \tau'$ *then* $\tau = \tau'$ *and* $r = r'$*.*

Proof. Suppose that (A) $\Gamma \mathbin{@} c \vdash e : \tau$ and (B) $\Gamma \mathbin{@} c' \vdash e : \tau'$. We show by induction over the typing derivation of $\Gamma \mathbin{@} c \vdash e : \tau$ that $\tau = \tau'$ and $c = c'$.

Case (*abs*): $e = \lambda x{:}\tau_1.e_1$. Then $\tau = \tau_1 \xrightarrow{c} \tau_2$ for some τ_2 and $\Gamma, x{:}\tau_1 \mathbin{@} c \vdash e : \tau_2$. By case (5) of Lemma 4, the final rule of the derivation (B) must have also been (*abs*) and this derivation has a sub-derivation with a conclusion $\Gamma, x{:}\tau_1 \mathbin{@} c' \vdash e : \tau_2'$. By the induction hypothesis $\tau_2 = \tau_2'$ and $c = c'$ and therefore also $\tau = \tau'$.

Cases (*var*), (*const*) are direct consequence of Lemma 4.

Cases (*app*) and (*let*) similarly to (*abs*). □

4.4 SYNTACTIC 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 coeffect systems. 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 dataflow is more complex.

4.4.1 *Syntactic properties*

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

$$
\begin{array}{rcll}
(\lambda x.e_1)\ e_2 & \longrightarrow_{\mathtt{cbn}} & e_1[x \leftarrow e_2] & (\textit{call-by-name}) \\
(\lambda x.e_1)\ v & \longrightarrow_{\mathtt{cbv}} & e_1[x \leftarrow v] & (\textit{call-by-value}) \\
e & \longrightarrow_{\eta} & \lambda x.e\ x & (\eta\textit{-expansion})
\end{array}
$$

The rules capture syntactic reductions that can be performed in a general calculus, without any knowledge of the specific notion of context. If the reductions preserve the type of the expression (type preservation), then an operational semantics can be defined as a repeated application of the rules, together with additional domain-specific rules for each context-aware language, until a specified normal form (i. e. a value) is reached.

In the rest of the section, we briefly outline the interpretation of the three rules and then we focus on call-by-value (Section 4.4.2) and call-by-name (Section 4.4.3) in more detail.

The focus of this chapter is on the general coeffect system and so we do not discuss the domain-specific reduction rules for individual context-aware language. Some work on both operational and denotational semantics of general coeffect systems has been done by Brunel et al. [17] and Breuvart and Pagani [15]. We give formal semantics of implicit parameters and dataflow in Chapter 5 by translation to a simple functional programming language instead.)

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 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 means that certain coeffect systems support call-by-name evaluation strategy and could be embedded in a purely functional language (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, the notion of value is defined syntactically. For example, in the *Effect* language [125], 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 defer all context demands of the body e and may have immediate context demands. Thus we say that e is a value if it is a value in the usual sense *and* has no immediate context demands. We define this formally in Section 4.4.2.

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

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. The hard work of reducing a context-dependent term to a *value* has to be provided for each system. Syntactic values are defined in the usual way:

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

The syntactic form *SynVal* captures syntactic values, but a context-dependency-free value in coeffect calculus cannot be defined purely syntactically, because a function $(\lambda x.e)$ may still have context demands – for example a function $(\lambda x.\texttt{prev}\ x)$ has an immediate context demand 1 (requiring 1 past value of all variables in the context).

Definition 4. *An expression* e *is a* value, *written as* $val(e)$ *if it is a syntactic value, i. e.* $e \in SynVal$ *and it has no context-dependencies, i. e.* $\Gamma\,@\,\mathsf{use} \vdash e : \tau$.

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

Lemma 6 (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 demands, that is* $\Gamma\,@\,r \vdash e[x \leftarrow v] : \tau$.

Proof. By induction over the type derivation, using the fact that x and v are annotated with use and that variables are never removed from the set Γ in the flat coeffect calculus. □

The substitution lemma 6 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. This is captured by the *approximation* property:

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

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

The property holds for the three systems we consider – for implicit parameters, this is an equality; for liveness and dataflow (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 $e[x \leftarrow v]$, the type preservation theorem is stated as follows:

Theorem 7 (Type preservation for call-by-value)**.** *In a flat coeffect system satisfying the approximation property, that is* $r \wedge t \ \leqslant\ r \oplus t$*, 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 final step simplifies the coeffect annotation using the fact that use is a unit of $\circledast$. From Lemma 6, $e[x \leftarrow v]$ has the same coeffect annotation as e. As $r \wedge t \leqslant r \oplus t$, we can apply subcoeffecting:

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

Comparing the final conclusions of the above two typing derivations shows that the reduction preserves type and coeffect annotation. □

4.4.3 *Call-by-name evaluation*

When reducing the expression $(\lambda x.e_1)\ e_2$ using the call-by-name strategy, the sub-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 term in coeffect calculi, but it does preserve the typing in two interesting cases.

Typing is preserved for different reasons in two of our systems, so we briefly review the concrete examples. Then, we prove the substitution lemma for two special cases of flat coeffects (Lemma 8 and Lemma 9) and finally, we state the conditions under which typing preservation holds for flat coeffect calculi (Theorem 10).

DATAFLOW. Reducing an expression $(\lambda x.e_1)\ e_2$ to $e_1[x \leftarrow e_2]$ does not always preserve the type of the expression in dataflow languages. This case is similar to the example shown earlier with effectful computations. As a minimal example, consider the substitution of a context-dependent expression `prev` z for a variable y in a function $\lambda x.y$:

$$\begin{aligned} y{:}\tau_1, z{:}\tau_1 \mathbin{@} 0 &\vdash \lambda x.y : \tau_1 \xrightarrow{0} \tau_2 && \text{(before)} \\ z{:}\tau_1 \mathbin{@} 1 &\vdash \lambda x.\texttt{prev}\ z : \tau_1 \xrightarrow{1} \tau_2 && \text{(after)} \end{aligned}$$

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 demands – at runtime, the code does not actually access a previous value of the argument x. This fact cannot be captured by a flat coeffect type system, but it can be captured using the structural system discussed in Chapter 6.

IMPLICIT PARAMETERS. In dataflow, substituting `prev` x for a variable y in an expression $\lambda z.y$ changes the context demands attached to the type of the function. This is the case not just for the preferred unique typing derivation, but for all possible typings that can be obtained using the (*abs*) rule. However, this is not the case for all systems. Consider a substitution $\lambda x.y[y \leftarrow ?p]$ that substitutes an implicit parameter access $?p$ for a free variable y under a lambda:

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

The (*after*) judgement 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 a substitution $\lambda x.y[y \leftarrow e]$ that substitutes an arbitrary expression e of type τ_1 with coeffects r for a variable y:

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

In the original expression, both the overall context and the function type are annotated with L, because the body contains a variable access. An expres-

sion e can always be treated as being annotated with L (because L is the top element of the lattice) and so we can also treat e as being annotated with coeffects L. As a result, substitution does not change the coeffect.

A GRAND CBN REDUCTION THEOREM. The above examples (implicit parameters and liveness) demonstrate two particular kinds of coeffect algebra for which call-by-name reduction preserves typing. Proving the type preservation separately provides more insight into how the systems work. We consider the two cases separately, but find a more general formulation for both of them.

Definition 5. *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 dataflow are examples of bottom-pointed coeffects. For top-pointed flat coeffects, the substitution lemma holds without additional demands:

Lemma 8 (Top-pointed substitution)**.** *In a top-pointed flat coeffect calculus with an algebra* $(\mathcal{C}, \circledast, \oplus, \wedge, \mathsf{use}, \mathsf{ign}, \leqslant)$*, when we substitute an expression* e_s *with arbitrary coeffects* s *for a variable* x *in* e_r*, the resulting expression is still typeable in a context with the original coeffect of* e_r*:*

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

Proof. Using subcoeffecting ($s \leqslant \mathsf{use}$) and a variation of Lemma 6. □

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

In a bottom pointed coeffect system, substituting e for x increases the context demands. However, if the system satisfies the strong condition that $\wedge = \circledast = \oplus$ then the context demands arising from the substitution can be associated with the context Γ, leaving the context demands 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 a set union) and for other coeffect systems that track sets of context demands discussed in Section 3.2.2. It allows the following substitution lemma:

Lemma 9 (Bottom-pointed substitution)**.** *In a bottom-pointed flat coeffect calculus with an algebra* $(\mathcal{C}, \circledast, \oplus, \wedge, \mathsf{use}, \mathsf{ign}, \leqslant)$ *where* $\wedge = \circledast = \oplus$ *is an idempotent and commutative operation and* $r \leqslant r' \Rightarrow \forall s.r \circledast s \leqslant r' \circledast s$ *then:*

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

Proof. By induction over $\vdash$, using the idempotent, commutative monoid structure to keep s with the free-variable context. See Appendix B.1. □

The flat system discussed here is *flexible enough* to let us always re-associate new context demands (arising from the substitution) with the free-variable context. In contrast, the structural system discussed in Chapter 6 is *precise*

enough to keep the coeffects associated with individual variables, thus preserving typing in a complementary way.

The two substitution lemmas discussed above 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 theorem holds:

Theorem 10 (Type preservation for call-by-name)**.** *In a coeffect system that satisfies the conditions for Lemma 8 or Lemma 9, if* $\Gamma \mathbin{@} r \vdash e : \tau$ *and* $e \to_{\mathsf{cbn}} e'$ *then it is also the case that* $\Gamma \mathbin{@} r \vdash e' : \tau$.

Proof. For top-pointed coeffect algebra (using Lemma 8), the proof is similar to that of Theorem 7, 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 $(\lambda x.e_r)\, e_s$ before reduction:

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

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 9 is:

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

From the assumptions of Lemma 9, 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 dataflow.

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 dataflow, the immediate coeffects are made larger by the lambda abstraction – the context demands of the function value are imposed on the declaration site of the new lambda abstraction. We remedy this limitation in the next chapter.

However, η-expansion preserves the type for implicit parameters and, more generally, for any flat coeffect algebra where $\oplus \;=\; \wedge$. Assuming $\to_\eta$ performs an expansion that turns a function-typed term e to a syntactic function $\lambda x.e\ x$, the following theorem holds:

Theorem 11 (Type preservation of 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* $\Gamma \mathbin{@} r \vdash e : \tau_1 \xrightarrow{s} \tau_2$ *and* $e \to_\eta e'$ *then* $\Gamma \mathbin{@} r \vdash e' : \tau_1 \xrightarrow{s} \tau_2$.

Proof. The following derivation shows that $\lambda x.f\ x$ has the same type as f:

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

The derivation starts with the expression e and derives the type for $\lambda x.e\ x$. The application yields context demands $r \oplus s$. In order to recover the original

$$(\textit{sub-trans})\ \frac{\tau_1 <: \tau_2 \qquad \tau_2 <: \tau_3}{\tau_1 <: \tau_3}$$

$$(\textit{sub-fun})\ \frac{\tau_1' <: \tau_1 \qquad \tau_2 <: \tau_2' \qquad r' \geqslant r}{\tau_1 \xrightarrow{r} \tau_2 <: \tau_1' \xrightarrow{r'} \tau_2'}$$

$$(\textit{sub-refl})\ \frac{}{\tau <: \tau}$$

Figure 24: Subtyping rules for flat coeffect calculus

typing, this must be equal to $r \wedge s$. The derivation shows just one possible typing – the expression $\lambda x.e\ x$ has other types – but this suffices. □

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. These include liveness and implicit parameters. Moreover, for implicit parameters the η-expansion holds as well, giving us both local soundness and local completeness as coined by Pfenning and Davies [87].

4.5 SYNTACTIC PROPERTIES AND EXTENSIONS

The flat coeffect algebra introduced in Section 4.2 requires a number of axioms. The axioms are required for three reasons – to be able to define the categorical structure in Section 5.2, 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 turn to the last point. We consider subcoeffecting and subtyping (Section 4.5.1), discuss what syntactic equivalences are permitted by the properties of $\wedge$ (Section 4.5.3) and we extend the calculus with pairs and units and discuss their syntactic properties (Section 4.5.4).

4.5.1 *Subcoeffecting and subtyping*

The *flat coeffect algebra* includes the $\leqslant$ relation which captures the ordering of coeffects and can be used to define subcoeffecting. Syntactically, an expression with context demands r' can be treated as an expression with a greater context. This is captured by the (*sub*) rule shown in Figure 22 (recall that for implicit parameters $\leqslant\ =\ \subseteq$):

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

Semantically, when read from the consequent to the assumption, this means that we can *drop* some of the provided context. For example, if an expression requires implicit parameters $\{?p\}$ it can be treated as requiring $\{?p, ?q\}$. The semantic function will then be provided with a dictionary containing both assignments and it can ignore (or even actively drop) the value for the unused parameter $?q$.

Subcoeffecting only affects the immediate coeffects attached to the free-variable context. In Figure 24, we add sub-typing on function types, making it possible to treat a function with smaller context demands as a function with greater context demands:

$$(\textit{sub-typ})\ \frac{\Gamma\ @\ r \vdash e : \tau \qquad \tau <: \tau'}{\Gamma\ @\ r \vdash e : \tau'}$$

	Derived	Definition	Simplified
Implicit parameters	$s_1 \cup (s_2 \cup r)$	$s \cup (s \cup r)$	$s \cup r$
Liveness	$s_1 \sqcap (s_2 \sqcup r)$	$s \sqcap (s \sqcup r)$	s
Dataflow	$max(s_1, s_2 + r)$	$max(s, s + r)$	$s + r$

Figure 25: Simplified coeffect annotation for let binding in three flat calculi instances

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 (*sub-typ*) rule allows using sub-typing on expressions in the coeffect calculus.

4.5.2 *Typing of let binding*

Recall the (*let*) rule in Figure 22. It annotates the expression `let` $x = e_1$ `in` e_2 with context demands $s \oplus (s \circledast r)$. This rule can be derived from the typing derivation for an expression $(\lambda x.e_2)\ e_1$ as a special case. We use the idempotence of $\wedge$ as follows:

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

This is one possible derivation, but other derivations may be valid for concrete coeffect systems. The design decision of using this particular derivation for the typing of `let` is motivated by the fact that the typing obtained using the special rule is more precise for all the examples considered in this chapter. To see this, assume an arbitrary splitting $s = s_1 \wedge s_2$. Figure 25 shows the coeffect annotation derived from $(\lambda x.e_2)\ e_1$, the coeffect annotation obtained by the (*let*) rule and the simplified coeffect annotation using the particular flat coeffect algebras.

It is perhaps somewhat unexpected that the annotation can be simplified in different ways for different examples. However, for all our systems, the simplified annotation (right column in Table 25) is more precise than the original (left column). Recall that $s = s_1 \wedge s_2$. The following holds:

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

In other words, the inequality states that using idempotence, we get a more precise typing. Using the $\geqslant$ operator of flat coeffect algebra, this property can be expressed in general as:

$$s_1 \oplus (s_2 \circledast r) \geqslant (s_1 \wedge s_2) \oplus ((s_1 \wedge s_2) \circledast r)$$

This property does not follow from the axioms 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 a reasonable basis for using the specialized (*let*) rule in the flat coeffect system.

4.5.3 *Properties of lambda abstraction*

In Section 4.1.1, we discussed how to reconcile two typings for lambda abstraction – for implicit parameters, the lambda function needs to split context demands using $r \cup s$, but for dataflow and liveness it suffices to duplicate the demand r of the body. We consider coeffect calculi for which the simpler duplication of coeffects is sufficient.

SIMPLIFIED ABSTRACTION. Recall that $(\mathcal{C}, \wedge)$ is a band, that is, $\wedge$ is idempotent and associative. The idempotence means that the context demands of the body can be required from both the declaration site and the call site. In Section 4.3.2, we introduced the (*idabs*) rule (repeated below for reference), which uses the idempotence and duplicates coeffect annotations:

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

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 following condition identifies coeffect calculi where (*abs*) can be derived from (*idabs*).

Definition 6. *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* $r \wedge s \leqslant r$.

Remark 12. *For a flat coeffect calculus with a strictly oriented algebra, equipped with subcoeffecting and subtyping, the standard (abs) rule can be derived from the (idabs) rule.*

Proof. The following derives the conclusion of (*abs*) using (*idabs*), subcoeffecting, sub-typing and the fact that the algebra is *strictly oriented*:

$$\dfrac{\dfrac{\dfrac{\Gamma, x{:}\tau_1 \mathbin{@} r \wedge s \vdash e : \tau_2}{\Gamma \mathbin{@} r \wedge s \vdash \lambda x.e : \tau_1 \xrightarrow{r \wedge s} \tau_2}\,(\textit{idabs})}{\Gamma \mathbin{@} r \vdash \lambda x.e : \tau_1 \xrightarrow{r \wedge s} \tau_2}\,(\textit{sub})\ (r \leqslant r \wedge s)}{\Gamma \mathbin{@} r \vdash \lambda x.e : \tau_1 \xrightarrow{s} \tau_2}\,(\textit{typ})\ (r \leqslant r \wedge s)$$

□

The practical consequence of Remark 12 is that, for strictly oriented coeffect calculi (such as our liveness and dataflow computations), the (*idabs*) rule not only determines a unique typing derivation (as discussed in Section 4.3.2), but it gives (together with subtyping and subcoeffecting) an equivalent type system.

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 13. *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*:*

$$\frac{\Gamma, x{:}\tau_1, y : \tau_2 \mathbin{@} r \wedge s \wedge t \vdash e : \tau_3}{\Gamma \mathbin{@} r' \vdash \lambda x.\lambda y.e : \tau_1 \xrightarrow{s'} (\tau_2 \xrightarrow{t'} \tau_3)}$$

$$(\textit{pair})\ \frac{\Gamma @ r \vdash e_1 : \tau_1 \qquad \Gamma @ s \vdash e_2 : \tau_2}{r \oplus s @ \Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

$$(\textit{proj})\ \frac{\Gamma @ r \vdash e : \tau_1 \times \tau_2}{\Gamma @ r \vdash \pi_i\ e : \tau_i}$$

$$(\textit{unit})\ \frac{}{\Gamma @ \mathsf{ign} \vdash () : \mathsf{unit}}$$

Figure 26: Typing rules for pairs and units

Intuitively, this means that the context demands of a function with multiple arguments can be split arbitrarily between the declaration site and (multiple) call sites.

4.5.4 *Language with pairs and unit*

To focus on the key aspects of flat coeffect systems, the calculus introduced in Section 4.2 consists only of variables, abstraction, application and let binding. Here, we extend it with pairs and the unit value to sketch how it can be turned into a more complete programming language and to further motivate the axioms for $\oplus$. The syntax of the language is extended as follows:

$$\begin{array}{lcl} e & ::= & \ldots \mid () \mid e_1, e_2 \\ \tau & ::= & \ldots \mid \mathsf{unit} \mid \tau \times \tau \end{array}$$

The typing rules for pairs and the unit value are shown in Figure 26. 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 *pointwise* operator $\oplus$. The operator models the case when the (same) available context is split and passed to two independent sub-expressions. Finally, the (*proj*) rule is uninteresting, because π_i can be viewed as a unary function.

PROPERTIES. Pairs and the unit value in a lambda calculus typically form a monoid up to isomorphism. Assuming $\simeq$ is an isomorphism that performs appropriate transformation on values, without affecting other properties (here, coeffects) of the expressions, the monoid axioms then correspond to the requirement that $(e_1, (e_2, e_3)) \simeq ((e_1, e_2), e_3)$ (associativity) and the demand that $((), e) \simeq e \simeq (e, ())$ (unit).

Thanks to the properties of $\oplus$, the flat coeffect calculus obeys the monoid axioms for pairs. 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 $\Gamma @ r \vdash e' : \tau$.

Remark 14. *For a flat coeffect calculus with pairs and units, the following holds:*

$$\begin{array}{rcll} \mathsf{assoc}\ (e_1, (e_2, e_3)) & \equiv & ((e_1, e_2), e_3) & \text{(associativity)} \\ \pi_1\ (e, ()) & \equiv & e & \text{(right unit)} \\ \pi_2\ ((), e) & \equiv & e & \text{(left unit)} \end{array}$$

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 constants in the language). □

The Remark 14 motivates the demand of the monoid structure $(\mathcal{C}, \oplus, \mathsf{ign})$ of the flat coeffect algebra. We require only unit and associativity axioms. In

our three examples, the $\oplus$ operator is also symmetric, which additionally guarantees that $(e_1, e_2) \simeq (e_2, e_1)$, which is a property that is expected to hold for λ-calculus.

4.6 SUMMARY

This chapter presented the *flat coeffect calculus* – a unified system for tracking *whole-context* properties of computations, that is properties related to the execution environment or the entire context in which programs 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 instantiated the system to capture three specific systems, namely liveness, dataflow and implicit parameters. However, the system is more general and can capture various other applications outlined in Section 3.2.

An inherent property of flat coeffect systems is the ambiguity of the typing for lambda abstraction. The body of a function requires certain context, but the context can be often provided by either the declaration site or the call site. Resolving this ambiguity has to be done differently for each concrete coeffect system, depending on its specific notion of context. We discussed this for implicit parameters, dataflow and liveness in Section 4.3 and noted that the result for dataflow and liveness generalizes for any coeffect calculus with strictly oriented coeffect algebra (Remark 12).

Finally, we introduced the equational theory for flat coeffect calculus. Although each concrete instance of flat coeffect calculus models a different notion of context, there are syntactic properties that hold for all flat coeffect systems satisfying certain additional conditions. In particular, two *type preservation* theorems prove that the operational semantics for two classes of flat coeffect calculi (including liveness and implicit parameters) can be based on standard call-by-name reduction.

In the next section, we move from abstract treatment of the flat coeffect calculus to a more concrete discussion. We explain its category-theoretical motivation, we use it to define translational semantics (akin to Haskell's "do" notation) and we prove a soundness result that well-typed programs in flat coeffect calculi for implicit parameters and dataflow do not get stuck in the translated version.

5 SEMANTICS OF FLAT COEFFECTS

The *flat coeffect calculus* introduced in the previous chapter uniformly captures a number of context-aware systems outlined in Chapter 3. The coeffect calculus can be seen as a *language framework* that simplifies the construction of concrete *domain-specific* coeffect languages. In the previous chapter, we discussed how it provides a type system that tracks the required context. In this chapter, we show that the language framework also provides a way for defining the semantics of concrete domain-specific coeffect languages, guides their implementation and simplifies safety proofs.

This is done using a *comonadically-inspired translation*. We translate a program written using the coeffect calculus into a simple functional language with additional coeffect-specific comonadically-inspired primitives that implement the concrete notion of context-awareness.

We use comonads in a syntactic way, following the example of Wadler and Thiemann [125] and Haskell's use of monads. The translation is the same for all coeffect languages, but the safety depends on the concrete coeffect-specific comonadically-inspired primitives. We prove the soundness of two concrete coeffect calculi (dataflow and implicit parameters). We note that the proof crucially relies on a relationship between coeffect annotations (provided by the type system) and the comonadically-inspired primitives (defining the semantics), which makes it easy to extend it to other concrete context-aware languages.

CHAPTER STRUCTURE AND CONTRIBUTIONS

- We introduce *indexed comonads*, a generalization of comonads, a category-theoretical dual of monads (Section 5.2) and we discuss how they provide semantics for coeffect calculus. This provides an insight into how (and why) the coeffect calculus works and shows an intriguing link with effects and monads.
- We use indexed comonads to guide our *translational semantics* of coeffect calculus (Section 5.3). We define a simple sound functional programming language (with type system and operational semantics). We extend it with uninterpreted comonadically-inspired primitives and define a translation that turns well-typed context-aware coeffect programs into programs of our functional language.
- For two sample coeffect calculi discussed earlier (dataflow and implicit parameters), we give reduction rules for the comonadically-inspired primitives and we extend the progress and preservation proofs, showing that well-typed programs produced by translation from two coeffect languages do not go wrong (Section 5.4)
- We note that the proof for concrete coeffect language (dataflow and implicit parameters) can be generalized – rather than reconsidering progress and preservation of the whole target language, we rely just on the correctness of the coeffect-specific comonadically-inspired primitives and abstraction mechanism provided by languages such as ML and Haskell (Section 5.6).

5.1 INTRODUCTION AND SAFETY

This chapter links together a number of different technical developments presented in this thesis. We take the flat coeffect calculus introduced in Chapter 4, define its *abstract comonadic semantics* and use it to define a translation that gives a *concrete operational semantics* to a number of concrete context-aware languages. The type system is used to guarantee that the resulting programs are correct. Finally, the development in this chapter is closely mirrored by the implementation presented in Chapter 7, which implements the translation together with an interpreter for the target language.

The key claim of this thesis is that writing context-aware programs using coeffects is easier and less error-prone. In this chapter, we substantiate the claim by showing that programs written in the coeffect calculus and evaluated using the translation provided here do not "go wrong".

To provide an intuition, consider two context-aware programs. The first calls a function that adds two implicit parameters in a context where one of them is defined. The second calculates the difference between the current and the previous value in a dataflow computation. For comparison, we show the code written in a coeffect dataflow language (on the left) and using standard ML-like libraries (on the right):

```
let add = fun x' → '                 let add = fun x params →
  ?one + ?two in                       lookup "one" params +
let ?one = 10 in                         lookup "two" params in
add 0                                add 0 (cons "one" 10 params)

let diff = fun x →                   let diff = fun x →
  x − prev x                           List.head x − List.head (List.tail x)
```

The add function (on the left) has a type int $\xrightarrow{\{?one,?two\}}$ int. We call it in a context containing ?one and so the coeffect of the program is {?two}. The safety property for implicit parameters (Theorem **??**) guarantees that, when executed in a context that provides a value for the implicit parameter ?one, the program reduces to a value of the correct type (or never terminates).

If we wrote the code without coeffects (on the right), we could use a dynamic map to pass around a dictionary of parameters (the lookup function obtains a value and add adds a new assignment to the map). In that case, the type of add is just int → int and so the user does not know which implicit parameters it will need.

Similarly, the diff function can be implemented in terms of lists (on the right) as a function of type num list → num. The function fails for input lists containing only zero or one elements and this is not reflected in the type and is not enforced by the type checker.

Using coeffects (on the left), the function has a type num $\xrightarrow{1}$ num meaning that it requires one past value (in addition to the current value). The safety property for dataflow (Theorem **??**) shows that, when called with a context that contains the required number of past values as captured by the coeffect type system, the function does not get stuck.

In summary, a coeffect type system, captures certain runtime demands of context-aware programs and (as we show in this chapter), eliminates common errors related to working with context.

5.2 CATEGORICAL MOTIVATION

The type system of the flat coeffect calculus arises syntactically, 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 translation discussed in Section 5.3.

5.2.1 *Comonads are to coeffects what monads are to effects*

The development in this chapter closely follows the example of effectful computations. Effect systems provide a type system for tracking effects and monadic translation can be used as a basis for implementing effectful domain-specific languages (e.g. through the "do" notation in Haskell).

The correspondence between effect system and monads has been pointed out by Wadler and Thiemann [125] and further explored by Atkey [6] and Vazou and Leijen [72]). This line of work relates effectful functions $\tau_1 \xrightarrow{\sigma} \tau_2$ to monadic computations $\tau_1 \to M^{\sigma}\tau_2$. In this chapter, we show a similar correspondence between *coeffect systems* and *comonads*. However, due to the asymmetry of λ-calculus, defining the semantics in terms of comonadic computations is not a simple mechanical dualisation of the work on effect systems and monads.

Our approach is inspired by the work of Uustalu and Vene [114] who present the semantics of contextual computations (mainly for dataflow) in terms of comonadic functions $C\tau_1 \to \tau_2$. We introduce *indexed comonads* that annotate the structure with information about the required context, i. e. $C^{r}\tau_1 \to \tau_2$. This is similar to the recent development on monads and effects by Katsumata [52] who parameterizes monads in a similar way to our indexed comonads.

5.2.2 *Categorical semantics*

As discussed in Section 2.3, a categorical semantics interprets terms as morphisms in some category. For typed calculi, the semantics defined by $[\![-]\!]$ usually interprets a term with a typing derivation leading to a judgement $x_1 : \tau_1 \ldots x_n : \tau_n \vdash e : \tau$ as a morphism $[\![\tau_1 \times \ldots \times \tau_n]\!] \to [\![\tau]\!]$.

For a well-defined semantics, we need to ensure that a well-typed term is assigned exactly one meaning. This can be achieved in a number of ways. First, we can prove the *coherence* [37] and show the morhphisms assigned to multiple typing derivations are equivalent. Second, the typing judgement can have a unique typing derivation. We follow the latter approach, using the unique typing derivation specified in Section 4.3.

As a best known example, Moggi [67] showed that the semantics of various effectful computations can be captured uniformly using (*strong*) *monads*. 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 failures (the Maybe monad), $M\alpha = \mathcal{P}(\alpha)$ models non-determinism (list monad) and side-effects can be modelled using $M\alpha = S \to (\alpha \times S)$ (state monad). Here, the structure of a strong monad provides necessary "plumbing" for composing monadic computations – sequential composition and strength for lifting free variables into the body of computation under a lambda abstraction.

Following a similar approach to Moggi, Uustalu and Vene [114] showed that (*monoidal*) *comonads* uniformly capture the semantics of various kinds of context-dependent computations [114]. For example, dataflow computations over non-empty lists are modelled using the non-empty list comonad $\mathsf{NEList}\,\alpha = \alpha + (\alpha \times \mathsf{NEList}\,\alpha)$.

The monadic and comonadic model outlined above 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[1].

In the next section, we introduce *indexed comonads*, which provide a more precise analysis and let us model computations with context demands r as functions $C^r\tau_1 \to \tau_2$ using an *indexed comonad* C^r.

5.2.3 *Introducing comonads*

In category theory, *comonad* is a dual of *monad*. As already outlined in Chapter 2, we obtain a definition of a comonad by taking a definition of a monad and "reversing the arrows". More formally, one of the equivalent definitions of comonad looks as follows (repeated from Section 2.3):

Definition 7. *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 [114] 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* `inl` *and* `inr` *for constructors of the left and right cases, respectively. The type* NEList *forms a comonad together with the following* counit *and* cobind *mappings:*

1 This is an over-simplification as we can use e. g. stacks of monad transformers and model functions with two different effects using $\tau_1 \to M_1\,(M_2\,\tau_2)$. However, monad transformers require the user to define complex systems of lifting to be composable. Consequently, they are usually used for capturing different kinds of impurities (exceptions, non-determinism, state), but not for capturing fine-grained properties (e. g. a set of memory regions that may be accessed by a stateful computation).

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

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 operation 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 $\mathsf{cobind}\ \mathsf{counit}\ \mathsf{l} = \mathsf{l}$.

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 $\mathtt{inl}\ ()$. 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 functions $\mathsf{Maybe}\ \tau_1 \to \tau_2$. To use such a model, we need to generalize comonads to *indexed comonads*.

5.2.4 *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 element use represents a variable access. An indexed comonad is formed by a data type (object mapping) $\mathsf{C}^{\mathsf{r}}\alpha$ where the r (also called *annotation*) is a member of the set $\mathcal{C}$ and determines what context is required.

Definition 8. *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* $(\mathsf{C}^{\mathsf{r}}, \mathsf{counit}_{\mathsf{use}}, \mathsf{cobind}_{\mathsf{r},\mathsf{s}})$ *where:*

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

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

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

Rather than defining a single mapping C, we are now defining a family of mappings C^{r} indexed by elements of the monoid structure $\mathcal{C}$. Similarly, the $\mathsf{cobind}_{\mathsf{r},\mathsf{s}}$ operation is now 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 objects α, β (modeling types) as indices, writing $\mathsf{cobind}^{\alpha,\beta}_{\mathsf{r},\mathsf{s}}$. For the purpose of this thesis, it is sufficient to omit the superscripts and treat cobind just as a family of mappings (rather than natural transformations). When this does not introduce ambiguity, we also occasionally omit the subscripts.

The counit operation is not defined for all $r \in \mathcal{C}$, but only for the unit use. Nevertheless we continue to write $\mathsf{counit}_{\mathsf{use}}$, but this is merely for symmetry and as a useful reminder to the reader. Crucially, this means that the operation is defined only for special contexts.

If we look at the indices in the comonad axioms, 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.

COMPOSITION. The co-Kleisli category that models sequential composition is formed by the unit arrow (provided by counit) together with the (associative) composition operation that composes computations with contextual demands as follows:

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

The composition $\hat{\circ}$ best expresses the intention of indexed comonads. Given two functions with contextual demands r and s, their composition is a function that requires $r \circledast s$. The contextual demands 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* $1 * 1 = 1$*. The mapping* C^1 *is the mapping* C *of the underlying comonad. The operations* counit_1 *and* $\mathsf{cobind}_{1,1}$ *are defined by the operations* counit *and* cobind *of the comonad.*

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

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

The *indexed Maybe comonad* models the semantics of the liveness coeffect system discussed in Section 3.2.3, where $C^L \alpha = \alpha$ models a live context and $C^D \alpha = 1$ models a dead context which does not contain a value. The counit operation extracts a value from a live context. As in the direct model discussed in Chapter B, the cobind operation 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 counit $()$ is not defined. This is not a problem with indexed comonads, because live contexts are distinguished by the (type-level) coeffect annotation and counit only needs to be defined on live contexts.

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. We assume that, all implicit parameters have the type* num. *The data type* $C^r\alpha = \alpha \times (r \to \mathsf{num})$ *represents a value* α *together with a function that associates a parameter value* num *with every implicit parameter name in* $r \subseteq \mathsf{Id}$. *The cobind and counit operations are defined as:*

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

In the definition, we use the notation (a, g) for a pair containing a value of type α together with g, which is a function of type $r \to \mathsf{num}$. The counit operation takes a value and a function (with empty set as a domain), ignores the function and extracts the value. The cobind operation uses the restriction operation $g|_r$ to restrict the domain of g to implicit parameters r and s in oder to get implicit parameters required by the argument of f and by the resulting computation, respectively (i. e. semantically, it *splits* the available context capabilities). The function g passed to cobind is defined on $r \cup s$ and so the restriction is valid in both cases.

The structure of *indexed comonads* is sufficient to model sequential composition of computations that use a single variable (as discussed in Section 2.3). To model full λ-calculus with lambda abstraction and multiple-variable contexts, we need additional operations introduced in the next section.

5.2.5 *Flat indexed comonads*

Because of the asymmetry of λ-calculus (discussed in Section 3.1), the duality between monads and comonads does not lead us towards the additional structure required to model full λ-calculus. In comonadic computations, additional information is attached to the context. In application and lambda abstraction, the context is propagated differently than in effectful computations.

To model the effectful λ-calculus, Moggi [67] requires a *strong* monad which has an additional operation $\mathsf{strength} : \alpha \times M\beta \to M(\alpha \times \beta)$. This allows lifting of free variables into an effectful computation. In Haskell, strength can be expressed in the host language and so is implicit.

To model λ-calculus with contextual properties, Uustalu and Vene [114] require *lax semi-monoidal* comonad. This structure requires an additional monoidal operation:

$$m : C\alpha \times C\beta \to C(\alpha \times \beta)$$

The m operation is needed in the semantics of lambda abstraction. Semantically, it represents merging of contextual capabilities attached to the variable contexts 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 not only operations for *merging*, but also for *splitting* of contexts.

Definition 9. *Given a flat coeffect algebra* $(\mathcal{C}, \circledast, \oplus, \wedge, \mathsf{use}, \mathsf{ign}, \leqslant)$, *a* flat indexed comonad *is an indexed comonad over the monoid* $(\mathcal{C}, \circledast, \mathsf{use})$ *equipped with families of operations* $\mathsf{merge}_{r,s}$, $\mathsf{split}_{r,s}$ *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* $C^{r\oplus s}(\alpha \times \beta) \to C^r\alpha \times C^s\beta$

The $\mathsf{merge}_{r,s}$ operation is the most interesting one. Given two comonadic values with additional contexts specified by r and s, it combines them into a single value with additional context $r \wedge s$. The $\wedge$ operation often represents *greatest lower bound*. 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*[2].

SEMANTICS OF SUBCOEFFECTING. Although we do not include subcoeffecting in the core flat coeffect calculus, it is an interesting extension to consider. Semantically, subcoeffecting drops some of the available contextual capabilities (drops some of the implicit parameters or some of the past values). This can be modelled by adding a (family of) lifting operation(s):

- $\mathsf{lift}_{r',r}$ is a family of mappings $C^{r'}\,\alpha \to C^{r}\,\alpha$ for all r', r such that $r \leqslant r'$

The axioms of flat coeffect algebra do not, in general, require that $r \leqslant r \oplus s$ and $s \leqslant r \oplus s$, but the property holds for the three sample coeffect systems we consider. For systems with the above property, the split operation can be expressed in terms of lifting (subcoeffecting) as follows:

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

The map_r operation is the mapping on arrows 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 use the argument c twice, use lifting to throw away additional parts of the context and then transform the values in the context.

This alternative definition is valid for our examples, but we do not use it for three reasons. First, it requires making subcoeffecting a part of the core definition. Second, this would be the only place where our semantics uses a variable *twice* (in this case c). Note therefore that our use of an explicit split means that the structure required by our semantics does not need to provide variable duplication and our model could be embedded in linear or affine category. Finally, explicit split is similar to the definition that is needed for structural coeffects in Chapter 6 and it makes the connection between the two easier to see.

EXAMPLES. All the examples of *indexed comonads* discussed in Section 5.2.4 can be extended into *flat indexed comonads*. Note however that this cannot be done mechanically, because each example requires us to define additional operations, specific for the example.

Example 9 (Monoidal comonads)**.** *Just like indexed comonads generalize comonads, the additional structure of flat indexed comonads generalizes the symmetric semimonoidal comonads of Uustalu and Vene [114]. 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.*

2 The $\wedge$ and $\oplus$ operations are the greatest and least upper bounds in the liveness and dataflow examples, but not for implicit parameters. However, they remain useful as an informal analogy.

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

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

Without the indexing, the merge operations implements *zip* on Maybe values, returning a value 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 $\mathsf{D} = r \sqcap s$), but when the input is *live*, the operation can perform implicit subcoeffecting and drop one of the values.

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 comonadic data type* $\mathsf{C}^r\alpha$ *is* $\alpha \times (r \to \mathsf{num})$ *where* num *is the type of implicit parameter values. The additional operations are defined as:*

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

The split operation splits the tuple and restricts the function (representing available implicit parameters) to the required subsets. The merge operation is more interesting. It uses the $\uplus$ 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 dataflow 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 number of available 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:*

$$\mathsf{counit}_0\langle a_0\rangle = a_0 \qquad\qquad \mathsf{C}^n\alpha = \underbrace{\alpha \times \ldots \times \alpha}_{(n+1)-\text{times}}$$

$$\begin{aligned}
&\mathsf{cobind}_{m,n}\ f\langle a_0, \ldots a_{m+n}\rangle =\\
&\quad \langle f\langle a_0, \ldots, a_m\rangle, \ldots, f\langle a_n, \ldots, a_{m+n}\rangle\rangle
\end{aligned}$$

$$\begin{aligned}
&\mathsf{merge}_{m,n}(\langle a_0, \ldots, a_m\rangle, \langle b_0, \ldots, b_n\rangle) =\\
&\quad \langle (a_0, b_0), \ldots, (a_{min(m,n)}, b_{min(m,n)})\rangle
\end{aligned}$$

$$\begin{aligned}
&\mathsf{split}_{m,n}\langle (a_0, b_0), \ldots, (a_{max(m,n)}, b_{max(m,n)})\rangle =\\
&\quad (\langle a_0, \ldots, a_m\rangle, \langle b_0, \ldots, b_n\rangle)
\end{aligned}$$

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 f, which itself requires m past values. When combining two lists, $\mathsf{merge}_{m,n}$ behaves as *zip* and produces a list of length of the shorter argument. When splitting a list, $\mathsf{split}_{m,n}$ needs the maximum of the lengths.

The semantics is defined over a typing derivation:

$$\frac{}{[\![\Gamma\,@\,\mathsf{use} \vdash x_i : \tau_i]\!]} = \frac{}{\pi_i \circ \mathsf{counit}_{\mathsf{use}}} \quad (\textit{var})$$

$$\frac{}{[\![\Gamma\,@\,\mathsf{ign} \vdash n : \mathsf{num}]\!]} = \frac{}{\mathsf{const}\ n} \quad (\textit{num})$$

$$\frac{[\![\Gamma, x : \tau_1\,@\,r \wedge s \vdash e : \tau_2]\!] = \mathsf{f}}{[\![\Gamma\,@\,r \vdash \lambda x.e : \tau_1 \xrightarrow{s} \tau_2]\!] = \mathsf{f} \circ \mathsf{curry}\ \mathsf{merge}_{r,s}} \quad (\textit{abs})$$

$$\frac{\begin{array}{rcl}[\![\Gamma\,@\,r \vdash e_1 : \tau_1 \xrightarrow{t} \tau_2]\!] &=& \mathsf{f} \\ [\![\Gamma\,@\,s \vdash e_2 : \tau_1]\!] &=& \mathsf{g}\end{array}}{\begin{array}{rcl}[\![\Gamma\,@\,r \oplus (s \circledast t) \vdash e_1\ e_2 : \tau_2]\!] &=& \mathsf{app} \circ \mathsf{f} \times (\mathsf{cobind}_{s,t}\ \mathsf{g}) \circ \mathsf{split}_{r,s \circledast t} \\ &&\quad \circ\ \mathsf{map}_{r \oplus (s \circledast t)}\ \mathsf{dup}\end{array}} \quad (\textit{app})$$

Assuming the following auxiliary operations:

$$\begin{array}{rcl}\mathsf{map}_r\ \mathsf{f} &=& \mathsf{cobind}_{\mathsf{use},r}\ (\mathsf{f} \circ \mathsf{counit}_{\mathsf{use}}) \\ \mathsf{const}\ v &=& \lambda x.v \\ \mathsf{curry}\ \mathsf{f}\ x\ y &=& \lambda f.\lambda x.\lambda y.\mathsf{f}\ (x, y) \\ \mathsf{dup}\ x &=& (x, x) \\ \mathsf{f} \times \mathsf{g} &=& \lambda(x, y).(\mathsf{f}\ x, \mathsf{g}\ y) \\ \mathsf{app}\ (\mathsf{f}, x) &=& \mathsf{f}\ x\end{array}$$

Figure 27: Categorical semantics of the flat coeffect calculus

5.2.6 *Semantics of flat calculus*

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

As discussed in Section 4.3, different typing derivations of coeffect programs may have different meaning (e. g. when working with implicit parameters) and so the semantics is defined over a *typing derivation* rather than over an *term*. To assign a semantics to a term, we need to choose a particular typing derivation. The algorithm for choosing a unique typing derivation for our three systems has been defined in Section 4.3.

CONTEXTS AND TYPES. The modelling of contexts and functions generalizes the concrete examples discussed in Chapter 3. 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:

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

EXPRESSIONS. The definition of the semantics is shown in Figure 27. For consistency with earlier work [114, 75], the definitions use a point-free categorical notation. The semantics uses a number of auxiliary definitions that can be expressed in a Cartesian-closed category such as currying curry, value duplication dup, function pairing (given $f : A \to B$ and $g : C \to D$ then

$f \times g : A \times C \to B \times D$) and application app. We will embed the definitions in a simple programming language later (Section 5.3).

The semantics of variable access and abstraction are the same as in the semantics of Uustalu and Vene [114], modulo the indexing. The semantics of variable access (*var*) uses $\mathsf{counit}_{\mathsf{use}}$ to extract a product of free variables, followed by projection π_i to obtain the variable value. Abstraction (*abs*) is interpreted as a curried function that takes the declaration site context and a function argument, merges them using $\mathsf{merge}_{r,s}$ and passes the result to the semantics of the body f. 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 (*app*) 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 using the semantic function f. The result is a function $C^t \tau_1 \to \tau_2$. The second context (with index $s \circledast t$) is used to evaluate e_2 and using the semantic function g and wrap it with context required by the function e_1 by applying $\mathsf{cobind}_{s,t}$. The app operation than applies the function (first element) on the argument (second element). Finally, numbers (*num*) become constant functions that ignore the context.

PROPERTIES. The categorical semantics in Section 5.3 defines a translation that embeds context-dependent computations in a functional programming language, similarly to how monads and the "do" notation provide a way of embedding effectful computations in Haskell.

An important property of the translation is that it respects the coeffect annotations provided by the type system. The annotations of the semantic functions match the annotations in the typing judgement and so the semantics is well-defined. This provides a further validation for the design of the type system developed in Section 4.2.2 – if the coeffect annotations for (*app*) and (*abs*) were different, we would not be able to provide a well-defined semantics using flat indexed comonads.

Informally, the following states that if we see the semantics as a translation, the resulting code is well-typed. We revisit the property in Lemma 22 once we define the target language and its typing.

Lemma 15 (Correspondence)**.** *In the semantics defined in Figure 27, the context annotations r of typing judgements $\Gamma @ r \vdash e : \tau$ and function types $\tau_1 \xrightarrow{r} \tau_2$ on the left-hand side correspond to the indices of mappings C^r in the corresponding semantic function on the right-hand side.*

Proof. By analysis of the semantic rules in Figure 27. We need to check that the domains and codomains of the morphisms in the semantics (right-hand side) match. □

Thanks to indexing, the correspondence provides more guarantees than for a non-indexed system. In the semantics, we not only know which values are comonadic, but we also know what contextual information they are required to provide. In Section 5.5, we note that this lets us generalize the proofs about concrete languages discussed in this chapter to a more general setting.

The semantics is also a generalization of the concrete semantics given when introducing context-aware programming languages in Chapter 3.

LANGUAGE SYNTAX

$$
\begin{array}{lcl}
v & = & n \mid \lambda x.e \mid (v_1, \ldots, v_n) \\
e & = & x \mid n \mid \pi_i\ e \mid (e_1, \ldots, e_n) \mid e_1\ e_2 \mid \lambda x.e \\
\tau & = & \mathsf{num} \mid \tau_1 \times \ldots \times \tau_n \mid \tau_1 \to \tau_2 \\
K & = & (v_1, \ldots, v_{i-1}, \_, e_{i+1}, \ldots e_n) \mid v\ \_ \mid \_\ e \mid \pi_i\ \_
\end{array}
$$

REDUCTION RULES

$$
\begin{array}{ll}
(fn) & (\lambda x.e)\ v \rightsquigarrow e[x \leftarrow v] \\
(prj) & \pi_i(v_1, \ldots, v_n) \rightsquigarrow v_i \\
(ctx) & K[e] \rightsquigarrow K[e'] \qquad\qquad (\text{when } e \rightsquigarrow e')
\end{array}
$$

TYPING RULES

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

$$
(num)\ \frac{}{\Gamma \vdash n : \mathsf{num}}
$$

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

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

$$
(proj)\ \frac{\Gamma \vdash e : \tau_1 \times \ldots \tau_i \times \ldots \times \tau_n}{\Gamma \vdash \pi_i\ e : \tau_i}
$$

$$
(tup)\ \frac{\forall i \in \{1 \ldots n\}.\ \Gamma \vdash e_i : \tau_i}{\Gamma \vdash (e_1, \ldots, e_n) : \tau_1 \times \ldots \times \tau_n}
$$

Figure 28: Common syntax and reduction rules of the target language

Theorem 16 (Generalization). *Consider a typing derivation obtained according to the rules for finding unique typing derivations as specified in Section 4.3 for a coeffect language with liveness, dataflow or implicit parameters.*

The semantics obtained by instantiating the rules in Figure 27 with the concrete operations defined in Example 10, Example 11 or Example 12 is the same as the one defined in Figure 15, Figure 10 and Figure 17, respectively.

Proof. Expansion of the definitions, using the unique typing derivation for dataflow and liveness and any typing derivation for implicit parameters. □

5.3 TRANSLATIONAL SEMANTICS

Although the notion of indexed comonads presented in the previous section is novel and interesting in its own, the main reason for introducing it is that we can view it as a translation that provides embedding of context-aware domain-specific languages in a simple target functional language. In this section, we follow the example of effects and monads and we use the semantics to define a translation akin to the "do" notation in Haskell.

A context-aware *source* program written using a concrete context-aware domain-specific language (capturing dataflow, implicit parameters or other kinds of context awareness) with domain-specific language extensions (the `prev` keyword, or the `?impl` syntax) is translated to a *target* language that

is not context-aware. The target language is a small functional language consisting of:

- Simple functional subset formed by lambda calculus with support for tuples and numbers.
- Comonadically-inspired primitives corresponding to *counit*, *cobind* and other operations of flat indexed comonads.
- Additional primitives that model contextual operations of each concrete coeffect language (*prev* for the `prev` keyword, *lookup* for the `?p` syntax and *letimpl* for the `let ?p = ...` notation).

The syntax, typing and reduction rules of the first part (simple functional language) are common to all concrete coeffect domain-specific languages. The syntax and typing rules of the second part (comonadically-inspired) primitives are also shared by all coeffect DSLs, however the *reduction rules* for the comonadically-inspired primitives differ – they capture the concrete notions of context. Finally, the third part (domain-specific primitives) will differ for each coeffect domain-specific language.

5.3.1 *Functional target language*

The target language for the translation is a simply typed lambda calculus with integers and tuples. We include integers as an example of a concrete type. Tuples are needed by the translation, which keeps a tuple of variable assignments. Encoding those without tuples would be possible, but cumbersome. In this section, we define the common parts of the language without the comonadically-inspired primitives.

The syntax of the target programming language is shown in Figure 28. The values include numbers n, tuples and function values. The expressions include variables x, values, lambda abstraction and application and operations on tuples. We do not need recursion or other data types (although a realistic programming language would include them). In what follows, we also use the following syntactic sugar for let binding:

$$\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \quad = \quad (\lambda x.e_2)\ e_1$$

Finally, $K[e]$ defines the syntactic evaluation context in which sub-expressions are evaluated. Together with the evaluation rules shown in Figure 28, this captures the standard call-by-name semantics of the common parts of the target language. The (standard) typing rules for the common expressions of the target language are also shown in Figure 28.

5.3.2 *Safety of functional target language*

The functional subset of the language described so far models a simple ML-like language. We choose call-by-value over call-by-name for no particular reason and Haskell-like language would work equally well.

The subset of the language introduced so far is type-safe in the standard sense that "well-typed programs do not get stuck". Although standard, we outline the important parts of the proof for the functional subset here, before we extend it to concrete context-aware languages in Section 5.4.

We use the standard syntactic approach to type safety introduced by Milner [66]. Following Wright, Felleisen and Pierce [88, 127], we prove the type preservation property (reduction does not change the type of an expression)

and the progress property (a well-typed expression is either a value or can be further reduced).

Lemma 17 (Canonical forms). *For all* e, τ, *if* $\vdash e : \tau$ *and* e *is a value then:*

1. *If* $\tau = \mathsf{num}$ *then* $e = n$ *for some* $n \in \mathbb{Z}$
2. *If* $\tau = \tau_1 \to \tau_2$ *then* $e = \lambda x.e'$ *for some* x, e'
3. *If* $\tau = \tau_1 \times \ldots \times \tau_n$ *then* $e = (v_1, \ldots, v_n)$ *for some* v_i

Proof. For (1), the last typing rule must have been (*num*); for (2), it must have been (*abs*) and for (3), the last typing rule must have been (*tup*) □

Lemma 18 (Preservation under substitution). *For all* $\Gamma, e, e', \tau, \tau'$, *if* $\Gamma, x : \tau \vdash e : \tau'$ *and* $\Gamma \vdash e' : \tau$ *then* $\Gamma \vdash e[x \leftarrow e'] : \tau$.

Proof. By induction over the derivation of $\Gamma, x : \tau \vdash e : \tau'$. □

Theorem 19 (Type preservation). *If* $\Gamma \vdash e : \tau$ *and* $e \to e'$ *then* $\Gamma \vdash e' : \tau$

Proof. Rule induction over $\rightsquigarrow$.

Case (*fn*): $e = (\lambda x.e_0)\ v$, from Lemma 18 it follows that $\Gamma \vdash e_0[x \leftarrow v] : \tau$.

Case (*prj*): $e = \pi_i(v_1, \ldots, v_n)$ and so the last applied typing rule must have been (*tup*) and $\Gamma \vdash (v_1, \ldots, v_n) : \tau_1 \times \ldots \times \tau_n$ and $\tau = \tau_i$. After applying (*prj*) reduction, $e' = v_i$ and so $\Gamma \vdash e' : \tau_i$.

Case (*ctx*): By induction hypothesis, the type of the reduced sub-expression does not change and the last used rule in the derivation of $\Gamma \vdash e : \tau$ also applies on e' giving $\Gamma \vdash e' : \tau$. □

Theorem 20 (Progress). *If* $\vdash e : \tau$ *then either* e *is a value or there exists* e' *such that* $e \rightsquigarrow e'$

Proof. By rule induction over $\vdash$.

Case (*num*): $e = n$ for some n and so e is a value.

Case (*abs*): $e = \lambda x.e'$ for some x, e', which is a value.

Case (*var*): This case cannot occur, because e is a closed expression.

Case (*app*): $e = e_1\ e_2$ which is not a value. By induction, e_1 is either a value or it can reduce. If it can reduce, apply (*ctx*) reduction with context $\_\ e$. Otherwise consider e_2. If it can reduce, apply (*ctx*) with context $v\ \_$. If both are values, Lemma 17 guarantees that $e_1 = \lambda x.e_1'$ and so we can apply reduction (*fn*).

Case (*proj*): $e = \pi_i e_0$ and $\tau = \tau_1 \times \ldots \tau_n$. If e_0 can be reduced, apply (*ctx*) with context $\pi_i\ \_$. Otherwise from Lemma 17, we have that $e_0 = (v_1, \ldots, v_n)$ and we can apply reduction (*prj*).

Case (*tup*): $e = (e_1, \ldots, e_n)$. If all sub-expressions are values, then e is also a value. Otherwise, we can apply reduction using (*ctx*) with a context $(v_1, \ldots, v_{i-1}, \_, e_{i+1}, \ldots, e_n)$. □

Theorem 21 (Safety of functional target language). *If* $\Gamma \vdash e : \tau$ *and* $e \rightsquigarrow^* e'$ *then either* e' *is a value of type* τ *or there exists* e'' *such that* $e' \rightsquigarrow e''$ *and* $\Gamma \vdash e'' : \tau$.

Proof. Rule induction over $\rightsquigarrow^*$ using Theorem 19 and Theorem 20. □

LANGUAGE SYNTAX. Given $(\mathcal{C}, \circledast, \oplus, \wedge, \mathsf{use}, \mathsf{ign}, \leqslant)$, extend the programming language syntax with the following constructs:

$$\begin{array}{rcl} e & = & \ldots \mid \mathsf{cobind}_{s,r}\ e_1\ e_2 \mid \mathsf{counit}_{\mathsf{use}}\ e \mid \mathsf{merge}_{r,s}\ e \mid \mathsf{split}_{r,s}\ e \\ \tau & = & \ldots \mid C^r\tau \\ K & = & \ldots \mid \mathsf{cobind}_{s,r}\ \_\ e \mid \mathsf{cobind}_{s,r}\ v\ \_ \mid \mathsf{counit}_{\mathsf{use}}\ \_ \\ & & \mid \mathsf{merge}_{r,s}\ \_ \mid \mathsf{split}_{r,s}\ \_ \end{array}$$

TYPING RULES. Given $(\mathcal{C}, \circledast, \oplus, \wedge, \mathsf{use}, \mathsf{ign}, \leqslant)$, add the typing rules:

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

$$(\textit{cobind})\ \frac{\Gamma \vdash e_1 : C^r\tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : C^{r \circledast s}\tau_1}{\Gamma \vdash \mathsf{cobind}_{r,s}\ e_1\ e_2 : C^s\tau_2}$$

$$(\textit{merge})\ \frac{\Gamma \vdash e : C^r\tau_1 \times C^s\tau_2}{\Gamma \vdash \mathsf{merge}_{r,s}\ e : C^{r \wedge s}(\tau_1 \times \tau_2)}$$

$$(\textit{split})\ \frac{\Gamma \vdash e : C^{r \oplus s}(\tau_1 \times \tau_2)}{\Gamma \vdash \mathsf{split}_{r,s}\ e : C^r\tau_1 \times C^s\tau_2}$$

Figure 29: Comonadically-inspired extensions for the target language

5.3.3 *Comonadically-inspired translation*

In Section 5.2, we presented the semantics of the flat coeffect calculus in terms of indexed comonads. We treated the semantics as denotational – interpreting the meaning of a given typing derivation of a program in terms of category theory.

In this chapter, we use the same structure in a different way. Rather than treating the rules as *denotation* in categorical sense, we treat them as *translation* from a source domain-specific coeffect language into a target language with comonadically-inspired primitives described in the previous section.

LANGUAGE EXTENSION. Given a coeffect language with a flat coeffect algebra $(\mathcal{C}, \circledast, \oplus, \wedge, \mathsf{use}, \mathsf{ign}, \leqslant)$, we first extend the language syntax and typing rules with terms that correspond to the comonadically-inspired operations. This is done in the same way for all concrete coeffect domain-specific languages and so we give the common additional syntax, evaluation context and typing rules once in Figure 29. We consider examples later in Section 5.4.

The new type C^r represents an indexed comonad, which is left abstract for now. The additional expressions such as $\mathsf{counit}_{\mathsf{use}}$ and $\mathsf{cobind}_{r,s}$ correspond to the operations of indexed comonads. Note that the we embed the coeffect annotations into the target language – these are known when translating a term with a chosen typing derivation from a source language and they will be useful when proving that sufficient context (as specified by the coeffect annotations) is available.

The translation is defined over a typing derivation:

$$\frac{}{[\![\Gamma \mathbin{@} \mathsf{use} \vdash x_i : \tau_i]\!]} = \frac{}{\lambda ctx.\pi_i\ (\mathsf{counit}_{\mathsf{use}}\ ctx)} \quad (var)$$

$$\frac{}{[\![\Gamma \mathbin{@} \mathsf{ign} \vdash n : \mathsf{num}]\!]} = \frac{}{\lambda ctx.n} \quad (num)$$

$$\frac{[\![\Gamma, x_i : \tau_1 \mathbin{@} r \wedge s \vdash e : \tau_2]\!]}{[\![\Gamma \mathbin{@} r \vdash \lambda x_i.e : \tau_1 \xrightarrow{s} \tau_2]\!]} = \frac{\mathsf{f}}{\begin{array}{l}\lambda ctx.\lambda v.\\ \quad \mathsf{let}\ \mathsf{reassoc} = \lambda x.\\ \qquad (\pi_1(\pi_1\ x), \ldots, \pi_{i-1}(\pi_1\ x), \pi_2\ x)\\ \quad \mathsf{f}\ (\mathsf{map}_{r \wedge s}\ \mathsf{reassoc}\ (\mathsf{merge}_{r,s}\ (ctx, v)))\end{array}} \quad (abs)$$

$$\frac{\begin{array}{rcl}[\![\Gamma \mathbin{@} r \vdash e_1 : \tau_1 \xrightarrow{t} \tau_2]\!] &=& \mathsf{f}\\ [\![\Gamma \mathbin{@} s \vdash e_2 : \tau_1]\!] &=& \mathsf{g}\end{array}}{[\![\Gamma \mathbin{@} r \oplus (s \circledast t) \vdash e_1\ e_2 : \tau_2]\!]} = \frac{}{\begin{array}{l}\lambda ctx.\\ \quad \mathsf{let}\ ctx_0 = \mathsf{map}_{r \oplus (s \circledast t)}\ \mathsf{dup}\ ctx\\ \quad \mathsf{let}\ (ctx_1, ctx_2) = \mathsf{split}_{r,s \circledast t}\ ctx_0\\ \quad \mathsf{f}\ ctx_1\ (\mathsf{cobind}_{s,t}\ \mathsf{g}\ ctx_2)\end{array}} \quad (app)$$

Assuming the following auxiliary operations:

$$\begin{array}{rcl}\mathsf{map}_r\ \mathsf{f} &=& \mathsf{cobind}_{\mathsf{use},r}\ (\lambda x.\mathsf{f}\ (\mathsf{counit}_{\mathsf{use}}\ x))\\ \mathsf{dup} &=& \lambda x.(x, x)\end{array}$$

Figure 30: Translation from a flat DSL to a comonadically-inspired target language

Figure 29 defines the syntax and the typing rules, but it does not define the reduction rules. These – together with the values for a concrete notion of context – will be defined separately for each individual coeffect language.

CONTEXTS AND TYPES. The interpretation of contexts and types in the category now becomes a translation from types and contexts in the source language into the types of the target language:

$$\begin{array}{rcl}[\![x_1 : \tau_1, \ldots, x_n : \tau_n \mathbin{@} r]\!] &=& C^r([\![\tau_1]\!] \times \ldots \times [\![\tau_n]\!])\\ [\![\tau_1 \xrightarrow{r} \tau_2]\!] &=& C^r[\![\tau_1]\!] \to [\![\tau_2]\!]\\ [\![\mathsf{num}]\!] &=& \mathsf{num}\end{array}$$

Here, a context becomes a comonadically-inspired data type wrapping a tuple of variable values and a coeffectful function is translated into an ordinary function in the target language with a comonadically-inspired data type wrapping the input type.

EXPRESSIONS. The rules shown in Figure 30 define how expressions of the source language are translated into the target language. The rules are very similar to those shown earlier in Figure 27. The consequent is now written as source code in the target programming language rather than as composition of morphisms in a category. However, thanks to the relationship between λ-calculus and Cartesian closed categories, both interpretations are equivalent.

One change from Figure 27 is that we are now more explicit about the tuple that contains variable assignments. Previously, we assumed that the tuple is appropriately reassociated. For programming language translation and the implementation (discussed in Chapter 7), we perform the reassociation explicitly. We keep a flat tuple of variables, so given $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$, the tuple has a type $\tau_1 \times \ldots \times \tau_n$. In (*var*), we access a variable using π, but in (*abs*), the merge operation produces a tuple $(\tau_1 \times \ldots \times \tau_{i-1}) \times \tau_i$ that we turn into a flat tuple $\tau_1 \times \ldots \times \tau_{i-1} \times \tau_i$ using the assoc function.

PROPERTIES. The most important property of the translation is that it produces well-typed programs in the target language. This is akin to the correspondence property of the semantics discussed earlier (Theorem 15), but now it has more obvious practical consequences.

In Section 5.4, we will prove safety properties of well-typed programs in the target language. Thanks to the fact that the translation produces a well-typed program means that we are also proving safety of well-typed programs in the source context-aware languages.

Theorem 22 (Well-typedness of the translation)**.** *Given a typing derivation for a well-typed closed expression* $@\,r \vdash e : \tau$ *written in a context-aware programming language that is translated to the target language as (we write* ... *for the omitted part of the translation tree):*

$$\frac{[\![\ (\ldots)\ ]\!]}{[\![@\,r \vdash e : \tau]\!]} = \frac{(\ldots)}{f}$$

Then f *is well-typed, i. e. in the target language:* $\vdash f : [\![\Gamma @\, r]\!] \to [\![\tau]\!]$.

Proof. By rule induction over the derivation of the translation. Given a judgement $x_1 : \tau_1 \ldots x_n : \tau_n @\, c \vdash e : \tau$, the translation constructs a function of type $C^c([\![\tau_1]\!] \times \ldots \times [\![\tau_n]\!]) \to [\![\tau]\!]$.

Case (*var*): $c = \mathsf{use}$ and $\tau = \tau_i$ and so $\pi_i(\mathsf{counit}_{\mathsf{use}}\ ctx)$ is well-typed.

Case (*num*): $\tau = \mathsf{num}$ and so the body n is well-typed.

Case (*abs*): The type of ctx is $C^r(\ldots)$ and the type of v is $C^s\tau_1$, calling $\mathsf{merge}_{r,s}$ and reassociating produces $C^{r \wedge s}(\ldots)$ as expected by f.

Case (*app*): After applying $\mathsf{split}_{r,s \circledast t}$, the types of ctx_1, ctx_2 are $C^r(\ldots)$ and $C^{s \circledast t}(\ldots)$, respectively. g requires $C^s(\ldots)$ and so the result of $\mathsf{cobind}_{s,t}$ is $C^t\tau_1$ as required by f. □

5.4 SAFETY OF CONTEXT-AWARE LANGUAGES

The language defined in Figure 28 and Figure 29 provide a general structure that we now use to prove the safety of various context-aware programming languages based on the coeffect language framework. As examples, we consider a language for dataflow computations (Section 5.4.1) and for implicit parameters (Section 5.4.2). In both cases, we extend the progress and preservation theorems of the functional subset of the target language, but the approach can be generalized as discussed in Section 5.5.

As outlined in the table at the beginning of Part ii, we now covered the parts of the semantics that are shared by all context-aware languages. This includes the functional target language with comonadically-inspired uninterpreted type $C^r\tau$ and the syntax for comonadically-inspired uninterpreted primitives such as $\mathsf{cobind}_{s,r}$ and $\mathsf{counit}_{\mathsf{use}}$, together with their typing.

Using dataflow and implicit parameters as two examples, we now add the domain-specific extensions needed for a concrete context-aware programming language. This includes syntax for values and expressions of the comonad-inspired type $C^r\tau$ and reduction rules for the comonadically-inspired operations ($\mathsf{cobind}_{s,r}$, counit_{use}, etc.).

5.4.1 *Coeffect language for dataflow*

The types of the comonadically-inspired operations are the same for each concrete coeffect DSL, but each DSL introduces its own *values* of type $C^r\tau$ and also its own reduction rules that define how comonadically-inspired operations evaluate.

We first consider dataflow computations. As discussed earlier in the semantics of dataflow, the indexed comonad for a context with n past values carries $n+1$ values. When reducing translated programs, the comonadic values will not be directly manipulated by the user code. In a programming language, it could be seen as an *abstract data type* whose only operations are the comonadically-inspired ones defined earlier, together with an additional *domain-specific* operation that models the `prev` construct.

The Figure 31 extends the target language with syntax, typing rules, additional translation rule and reductions for modelling dataflow computations. We introduce a new kind of values written as $\mathsf{Df}\langle v_0, \ldots, v_n\rangle$ and a matching kind of expressions. We specify how the `prev` keyword is translated into a prev_r operation of the target language and we also add a typing rule (*df*) that checks the types of the elements of the stream and also guarantees that the number of elements in the stream matches the number in the coeffect. The additional reduction rules mirror the semantics that we discussed in Example 12 when discussing the indexed list comonad.

PROPERTIES. Now consider a target language consisting of the core (ML-subset) defined by the syntax, reduction rules and typing rules given in Figure 28 and comonadically-inspired primitives defined in Figure 29 and also concrete notion of comonadically-inspired value and reduction rules for dataflow as defined in Figure 31.

In order to prove type safety, we first extend the *canonical forms lemma* (Lemma 17) and the *preservation under substitution lemma* (Lemma 18). Those need to consider the new (*df*) and (*prev*) typing rules and substitution under the newly introduced expression forms $\mathsf{Df}\langle\ldots\rangle$ and prev_n. We show that the translation rule for `prev` produces well-typed expressions. Finally, we extend the type preservation (Theorem 19) and progress (Theorem 20) theorems.

Theorem 23 (Well-typedness of the `prev` translation)**.** *Given a typing derivation for a well-typed closed expression* $@\,r \vdash e : \tau$*, the translated program* f *obtained using the rules in Figure 30 and Figure 31 is well-typed, i. e. in the target language:* $\vdash f : [\![\Gamma\,@\,r]\!] \to [\![\tau]\!]$.

Proof. By rule induction over the derivation of the translation.

Case (*var, num, abs, app*): As before.

Case (*prev*): Type of *ctx* is $C^{n+1}\tau$ and so we can apply the (*prev*) rule. □

Lemma 24 (Canonical forms)**.** *For all* e, τ*, if* $\vdash e : \tau$ *and* e *is a value then:*

1. *If* $\tau = \mathsf{num}$ *then* $e = n$ *for some* $n \in \mathbb{Z}$
2. *If* $\tau = \tau_1 \to \tau_2$ *then* $e = \lambda x.e'$ *for some* x, e'

LANGUAGE SYNTAX

$$\begin{aligned} v &= \ldots \mid \mathsf{Df}\langle v_0, \ldots, v_n\rangle \\ e &= \ldots \mid \mathsf{Df}\langle e_0, \ldots, e_n\rangle \mid \mathsf{prev}_n\ e \\ K &= \ldots \mid \mathsf{prev}_n\ \_ \mid \mathsf{Df}\langle v_0, \ldots, v_{i-1}, \_, e_{i+1} \ldots, e_n\rangle \end{aligned}$$

TYPING RULES

$$(df)\ \frac{\forall i \in \{0 \ldots n\}.\ \Gamma \vdash e_i : \tau}{\Gamma \vdash \mathsf{Df}\langle e_0, \ldots, e_n\rangle : C^n\tau}$$

$$(prev)\ \frac{\Gamma \vdash e : C^{n+1}\tau}{\Gamma \vdash \mathsf{prev}_n\ e : C^n\tau}$$

TRANSLATION

$$\frac{[\![\Gamma\,@\,n+1 \vdash e : \tau]\!]}{[\![\Gamma\,@\,n \vdash \mathsf{prev}\ e : \tau]\!]} = \frac{f}{\lambda ctx.\mathsf{prev}_n\ ctx}$$

REDUCTION RULES

$$\begin{aligned} &(counit) && \mathsf{counit}_0(\mathsf{Df}\langle v_0\rangle) \rightsquigarrow v_0 \\ &(cobind) && \mathsf{cobind}_{m,n}\ f\ (\mathsf{Df}\langle v_0, \ldots v_{m+n}\rangle) \rightsquigarrow \\ &&& \quad (\mathsf{Df}\langle f(\mathsf{Df}\langle v_0, \ldots, v_m\rangle), \ldots, f(\mathsf{Df}\langle v_n, \ldots, v_{m+n}\rangle)\rangle) \\ &(merge) && \mathsf{merge}_{m,n}((\mathsf{Df}\langle v_0, \ldots, v_m\rangle), (\mathsf{Df}\langle v'_0, \ldots, v'_n\rangle)) \rightsquigarrow \\ &&& \quad (\mathsf{Df}\langle (v_0, b_0), \ldots, (v_{min(m,n)}, v'_{min(m,n)})\rangle) \\ &(split) && \mathsf{split}_{m,n}(\mathsf{Df}\langle (v_0, b_0), \ldots, (v_{max(m,n)}, b_{max(m,n)})\rangle) \rightsquigarrow \\ &&& \quad \mathsf{Df}\langle v_0, \ldots, v_m\rangle, (\mathsf{Df}\langle b_0, \ldots, b_n\rangle \\ &(prev) && \mathsf{prev}_n(\mathsf{Df}\langle v_0, \ldots, v_n, v_{n+1}\rangle) \rightsquigarrow \\ &&& \quad \mathsf{Df}\langle v_0, \ldots, v_n\rangle \end{aligned}$$

Figure 31: Additional constructs for modelling dataflow in the target language

3. *If* $\tau = \tau_1 \times \ldots \times \tau_n$ *then* $e = (v_1, \ldots, v_n)$ *for some* v_i
4. *If* $\tau = C^n\tau_1$ *then* $e = \mathsf{Df}\langle v_0, \ldots v_n\rangle$ *for some* v_i

Proof. (1,2,3) as before; for (4) the last typing rule must have been (*df*). □

Lemma 25 (Preservation under substitution). *For all* $\Gamma, e, e', \tau, \tau'$, *if* $\Gamma, x : \tau \vdash e : \tau'$ *and* $\Gamma \vdash e' : \tau$ *then* $\Gamma \vdash e[x \leftarrow e'] : \tau$.

Proof. By induction over the derivation of $\Gamma, x : \tau \vdash e : \tau'$ as before, with new cases for $\mathsf{Df}\langle \ldots \rangle$ and prev_n. □

Theorem 26 (Type preservation). *If* $\Gamma \vdash e : \tau$ *and* $e \rightsquigarrow e'$ *then* $\Gamma \vdash e' : \tau$

Proof. Rule induction over $\rightsquigarrow$.

Case (*fn*, *prj*, *ctx*): As before, using Lemma 25 for (*fn*).

Case (*counit*): $e = \mathsf{counit}_0(\mathsf{Df}\langle v_0\rangle)$. The last rule in the type derivation of e must have been (*counit*) with $\Gamma \vdash \mathsf{Df}\langle v_0\rangle : C^0\tau$ and therefore $\Gamma \vdash v_0 : \tau$.

Case (*cobind*): $e = \mathsf{cobind}_{m,n}\ f\ (\mathsf{Df}\langle v_0, \ldots v_{m+n}\rangle)$. The last rule in the type derivation of e must have been (*cobind*) with a type $\tau = C^n\tau_2$ and as-

sumptions $\Gamma \vdash f : C^{m}\tau_1 \to \tau_2$ and $\Gamma \vdash \mathsf{Df}\langle v_0, \ldots v_{m+n}\rangle : C^{m+n}\tau$. The reduced expression has a type $C^{n}\tau_2$:

$$\frac{\Gamma \vdash f : C^{m}\tau_1 \to \tau_2 \qquad \forall i \in 0 \ldots n.\ \Gamma \vdash \mathsf{Df}\langle v_i, \ldots, v_{i+m}\rangle : C^{m}\tau_1}{\dfrac{\forall i \in 0 \ldots n.\ \Gamma \vdash f(\mathsf{Df}\langle v_i, \ldots, v_{i+m}\rangle) : \tau_2}{\Gamma \vdash \mathsf{Df}\langle f(\mathsf{Df}\langle v_0, \ldots, v_m\rangle), \ldots, f(\mathsf{Df}\langle v_n, \ldots, v_{m+n}\rangle)\rangle : C^{n}\tau_2}}$$

Case (*merge*, *split*, *next*): Similar. In all three cases, the last typing rule in the derivation of e guarantees that the stream contains a sufficient number of elements of correct type. □

Theorem 27 (Progress). *If $\vdash e : \tau$ then either e is a value or there exists e' such that $e \rightsquigarrow e'$*

Proof. By rule induction over $\vdash$.

Case (*num,abs,var,app,proj,tup*): As before, using the adapted canonical forms lemma (Lemma 24) for (*app*) and (*proj*).

Case (*counit*): $e = \mathsf{counit}_{\mathsf{use}}\ e_1$. If e_1 is not a value, it can be reduced using (*ctx*) with context $\mathsf{counit}_{\mathsf{use}}\ \_$, otherwise it is a value. From Lemma 24, $e_1 = \mathsf{Df}\langle v\rangle$ and so we can apply (*counit*) reduction rule.

Case (*cobind*): $e = \mathsf{cobind}_{m,n}\ e_1\ e_2$. If e_1 is not a value, reduce using (*ctx*) with context $\mathsf{cobind}_{m,n}\ \_\ e$. If e_2 is not a value reduce using (*ctx*) with context $\mathsf{cobind}_{m,n}\ v\ \_$. If both are values, then from Lemma 24, we have that $e_2 = \mathsf{Df}\langle v_0, \ldots v_{m+n}\rangle$ and so we can apply the (*cobind*) reduction.

Case (*merge*): $e = \mathsf{merge}_{m,n} e_1$. If e_1 is not a value, reduce using (*ctx*) with context $e = \mathsf{merge}_{m,n}\ \_$. If e_1 is a value, it must be a pair of streams $(\mathsf{Df}\langle v_0, \ldots, v_m\rangle, \mathsf{Df}\langle v'_0, \ldots, v'_n\rangle)$ using Lemma 24 and it can reduce using (*merge*) reduction.

Case (*df*): $e = \mathsf{Df}\langle e_0, \ldots, e_n\rangle$. If e_i is not a value then reduce using (*ctx*) with context $\mathsf{Df}\langle v_0, \ldots, v_{i-1}, \_, e_{i+1} \ldots, e_n\rangle$. Otherwise, $e_0, \ldots, e_n$ are values and so $\mathsf{Df}\langle e_0, \ldots, e_n\rangle$ is also a value.

Case (*split*, *prev*): Similar. Either sub-expression is not a value, or the type guarantees that it is a stream with correct number of elements to enable the (*split*) or (*prev*) reduction, respectively. □

Theorem 28 (Safety of context-aware dataflow language). *If $\Gamma \vdash e : \tau$ and $e \rightsquigarrow^* e'$ then either e' is a value of type τ or there exists e'' such that $e' \rightsquigarrow e''$ and $\Gamma \vdash e'' : \tau$.*

Proof. Rule induction over $\rightsquigarrow^*$ using Theorem 26 and Theorem 27. □

5.4.2 *Coeffect language for implicit parameters*

We now turn to our second example. As discussed earlier (Example 11), implicit parameters can be modelled by an indexed product comonad, which annotates a value with additional context – in our case, a mapping from implicit parameter names to their values. In this section, we embed this model into the target language.

As with dataflow computations, we take the core functional subset (Figure 28) with comonadically-inspired extensions (Figure 29) and we specify a new kind of values of type $C^{r}\tau$ and domain-specific reduction rules that specify how the operations propagate and access the context containing implicit parameter bindings. Again, the $C^{r}\tau$ values can be seen as an *abstract data type*, which are never manipulated directly, except by the comonadically-inspired operations ($\mathsf{cobind}_{s,r}$, $\mathsf{counit}_{\mathsf{use}}$, etc.).

DOMAIN-SPECIFIC EXTENSIONS. The Figure 32 shows extensions to the target language for modelling implicit parameters. A comonadic value with a coeffect $\{?p_1, \ldots, ?p_n\}$ is modelled by a new kind of value written as $\mathsf{Impl}(v, \{?p_1 \mapsto v_1, \ldots ?p_n \mapsto v_n\})$ which contains a value v together with implicit parameter assignments for all the parameters specified in the coeffect. We add a corresponding kind of expression with its typing rule (*impl*).

There are also two domain-specific operations for working with implicit parameters. The $\mathsf{lookup}_{?p}$ operation reads a value of an implicit parameter and the $\mathsf{letimpl}_{?p,r}$ operation adds a mapping assigning a value to an implicit parameter ?p. The typing rule (*lookup*) specifies that the accessed parameter need to be a part of the context and the rule (*letimpl*) specifies that the letimpl operation extends the context with a new implicit parameter binding.

The new translation rules specify how implicit parameter access, written as ?p, and implicit parameter binding, written as `let` ?p $= e_1$ `in` e_2 are translated to the target language. The first one is straightforward. The binding is similar to the translation for function application – we split the context, evaluate e_1 using the first part of the context ctx_1 and then add the new binding to the remaining context ctx_2.

Finally, Figure 32 also defines the reduction rules. The (*lookup*) rule accesses an implicit parameter and (*letimpl*) adds a new binding. The reduction rules closely model the product comonad discussed in Example 11. Reductions for (*cobind*) and (*split*) restrict the set of available implicit parameters according to the annotations and (*merge*) combines them, preferring the values from the call site.

For the semantics of implicit parameter programs that we consider, the preference of call site bindings over declaration site bindings in (*merge*) does not matter. The unique typing derivations for implicit parameter coeffects obtained in Section 4.3 always split implicit parameters into *disjoint sets*, so preferences do not come into play.

PROPERTIES. We now prove the type safety of of a context-aware programming language with implicit parameters. To do this, we prove safety of the target functional language with specific extensions for implicit parameters and we show that the translation from context-aware programming language with implicit parameters produces well-typed programs in the target language.

The target language consists of the core functional language subset (Figure 28) with the comonadically-inspired extensions (Figure 29) and the domain-specific extensions for implicit parameters defined in Figure 32. The well-typedness of the translation has been discussed earlier (Theorem 22) and we extend it to cover operations specific for implicit parameters below (Theorem 29).

As for dataflow computations, we prove the type safety by extending the preservation (Theorem 19) and progress (Theorem 20) for the core functional subset of the language, but it is worth noting that the key parts of the proofs are centered around the new reduction rules for comonadically-inspired primitives and newly defined Impl values. These do not interact with the rest of the language in any unexpected ways.

Theorem 29 (Well-typedness of the implicit parameters translation)**.** *Given a typing derivation for a well-typed closed expression* $@\,r \vdash e : \tau$*, the translated program* f *obtained using the rules in Figure 30 and Figure 32 is well-typed, i. e. in the target language:* $\vdash f : [\![\Gamma\, @\, r]\!] \to [\![\tau]\!]$.

Proof. By rule induction over the derivation of the translation.

LANGUAGE SYNTAX

$$
\begin{array}{rcl}
v & = & \ldots \mid \mathsf{Impl}(v, \{?p_1 \mapsto v_1, \ldots, ?p_n \mapsto v_n\}) \\
e & = & \ldots \mid \mathsf{Impl}(e, \{?p_1 \mapsto e_1, \ldots, ?p_n \mapsto e_n\}) \\
 & & \mid \mathsf{lookup}_{?p}\ e \mid \mathsf{letimpl}_{?p,r}\ e_1\ e_2 \\
K & = & \ldots \mid \mathsf{lookup}_{?p}\ \_ \mid \mathsf{letimpl}_{?p,r}\ \_\ e \mid \mathsf{letimpl}_{?p,r}\ v\ \_ \\
 & & \mid \mathsf{Impl}(\_, \{?p_1 \mapsto e_1, .., ?p_n \mapsto e_n\}) \\
 & & \mid \mathsf{Impl}(v, \{?p_1 \mapsto v_1, .., ?p_{i-1} \mapsto v_{i-1}, ?p_i \mapsto \_, ?p_{i+1} \mapsto v_{i+1}, ..?p_n \mapsto e_n\})
\end{array}
$$

TYPING RULES

$$
(\textit{impl})\ \frac{\Gamma \vdash e : \tau \qquad \forall i \in \{1 \ldots n\}.\ \Gamma \vdash e_i : \mathsf{num}}{\Gamma \vdash \mathsf{Impl}(e, \{?p_1 \mapsto e_1, \ldots, ?p_n \mapsto e_n\}) : C^{\{?p_1, \ldots, ?p_n\}}\tau}
$$

$$
(\textit{lookup})\ \frac{\Gamma \vdash e : C^{\{?p\}}\tau}{\Gamma \vdash \mathsf{lookup}_{?p}\ e : \mathsf{num}}
$$

$$
(\textit{letimpl})\ \frac{\Gamma \vdash e_1 : \mathsf{num} \qquad \Gamma \vdash e_2 : C^{\{?p_1, \ldots, ?p_n\}}\tau}{\Gamma \vdash \mathsf{letimpl}_{?p,\{?p_1, \ldots, ?p_n\}}\ e_1\ e_2 : C^{\{?p_1, \ldots, ?p_n, ?p\}}\tau}
$$

TRANSLATION

$$
(\textit{lookup})\ \frac{}{[\![\Gamma \circledast \{?p\} \vdash ?p : \mathsf{num}]\!]} = \frac{}{\lambda \mathsf{ctx}.\mathsf{lookup}_{?p}\ ctx}
$$

$$
(\textit{letimpl})\ \frac{[\![\Gamma \circledast r \vdash e_1 : \tau_1]\!] \quad = \quad f \qquad [\![\Gamma \circledast s \vdash e_2 : \tau_2]\!] \quad = \quad g}{[\![\Gamma \circledast r \cup (s \setminus \{?p\}) \vdash \mathsf{let}\ ?p = e_1\ \mathsf{in}\ e_2 : \tau_2]\!]}
$$

$$
= \lambda ctx.\ \mathsf{let}\ ctx_0 = \mathsf{map}_{r \cup (s \setminus \{?p\})}\ \mathsf{dup}\ ctx\ \mathsf{let}\ (ctx_1, ctx_2) = \mathsf{split}_{r,(s \setminus \{?p\})}\ ctx_0\ g\ (\mathsf{letimpl}_{?p,(s \setminus \{?p\})}\ (f\ ctx_1)\ ctx_2)
$$

REDUCTION RULES

$$
\begin{array}{ll}
(\textit{counit}) & \mathsf{counit}_{\emptyset}\ (\mathsf{Impl}(v, \ldots)) \rightsquigarrow v \\
(\textit{cobind}) & \mathsf{cobind}_{r,s}\ f\ (\mathsf{Impl}(v, \{?p_1 \mapsto v_1, \ldots, ?p_n \mapsto v_n\})) \rightsquigarrow \\
 & \quad \mathsf{Impl}(f\ (\mathsf{Impl}(v, \{?p_i \mapsto v_i \mid p_i \in r\})), \{?p_i \mapsto v_i \mid p_i \in s\}) \\
(\textit{merge}) & \mathsf{merge}_{r,s}(\ \mathsf{Impl}(v, \{?p_1 \mapsto v_1, \ldots, ?p_n \mapsto v_n\}) \\
 & \qquad \mathsf{Impl}(v', \{?p'_1 \mapsto v'_1, \ldots, ?p'_n \mapsto v'_n\})\ ) \rightsquigarrow \\
 & \quad \mathsf{Impl}((v, v'), \{?p_i \mapsto v_i \mid ?p \in r \setminus s\} \cup \{?p'_i \mapsto v'_i \mid ?p' \in s\}) \\
(\textit{split}) & \mathsf{split}_{r,s}(\mathsf{Impl}((v, v'), \{?p_1 \mapsto v_1, \ldots, ?p_n \mapsto v_n\})) \rightsquigarrow \\
 & \quad \mathsf{Impl}(v, \{?p_i \mapsto v_i \mid p_i \in r\}), \mathsf{Impl}(v', \{?p_i \mapsto v_i \mid p_i \in s\}) \\
(\textit{letimpl}) & \mathsf{letimpl}_{?p,r}\ v'\ (\mathsf{Impl}(v, \{?p_1 \mapsto v_1, \ldots, ?p_n \mapsto v_n\})) \rightsquigarrow \\
 & \quad \mathsf{Impl}(v, \{?p_i \mapsto v_i \mid ?p_i \in r, ?p_i \neq ?p\} \cup \{?p \mapsto v'\}) \\
(\textit{lookup}) & \mathsf{lookup}_{?p_i}(\mathsf{Impl}(v, \{?p_i \mapsto v_i\}) \rightsquigarrow v_i
\end{array}
$$

Figure 32: Additional constructs embedding implicit parameters into the language

Case (*var, num, abs, app*): As before.

Case (*lookup*): The type of *ctx* has a coeffect $\{?p\}$ which includes the parameter $?p$ as required in order to use the (*lookup*) typing rule.

Case (*letimpl*): The type of *ctx* matches with the input type of $\mathsf{map}_{r \cup (s \setminus \{?p\})}$. After duplication and splitting the context, ctx_1 and ctx_2 have types $C^r(\ldots)$ and $C^{s \setminus \{?p\}}(\ldots)$, respectively. This matches with the expected types of f and letimpl. The context returned by letimpl then matches the one required by g. □

Lemma 30 (Canonical forms). *For all* e, τ, *if* $\vdash e : \tau$ *and* e *is a value then:*

1. *If* $\tau = \mathsf{num}$ *then* $e = n$ *for some* $n \in \mathbb{Z}$
2. *If* $\tau = \tau_1 \to \tau_2$ *then* $e = \lambda x.e'$ *for some* x, e'
3. *If* $\tau = \tau_1 \times \ldots \times \tau_n$ *then* $e = (v_1, \ldots, v_n)$ *for some* v_i
4. *If* $\tau = C^{\{?p_1, \ldots, ?p_n\}}\tau_1$ *then* $e = \mathsf{Impl}(v, \{?p_1 \mapsto v_1, \ldots, ?p_n \mapsto v_n\})$

Proof. (1,2,3) as before; for (4) the last typing rule must have been (*impl*). □

Lemma 31 (Preservation under substitution). *For all* $\Gamma, e, e', \tau, \tau'$, *if* $\Gamma, x : \tau \vdash e : \tau'$ *and* $\Gamma \vdash e' : \tau$ *then* $\Gamma \vdash e[x \leftarrow e'] : \tau$.

Proof. By induction over the derivation of $\Gamma, x : \tau \vdash e : \tau'$ as before, with new cases for $\mathsf{Impl}(e, \{\ldots\})$, $\mathsf{lookup}_{?p}$ and $\mathsf{letimpl}_{?p,r}$. □

Theorem 32 (Type preservation). *If* $\Gamma \vdash e : \tau$ *and* $e \rightsquigarrow e'$ *then* $\Gamma \vdash e' : \tau$

Proof. Rule induction over $\rightsquigarrow$.

Case (*fn, prj, ctx*): As before, using Lemma 31 for (*fn*).

Case (*counit*): $e = \mathsf{counit}_0(\mathsf{Impl}(v, \{\,\}))$. The last rule in the type derivation of e must have been (*counit*) with $\Gamma \vdash \mathsf{Impl}(v, \{\,\}) : C^{\emptyset}\tau$ which, in turn, required that $\Gamma \vdash v : \tau$.

Case (*cobind*): $e = \mathsf{cobind}_{r,s}\ f\ (\mathsf{Impl}(v, \{?p_1 \mapsto v_1, \ldots, ?p_n \mapsto v_n\}))$. The last rule in the type derivation of e must have been (*cobind*) with a type $\tau = C^s\tau_2$ and assumptions $\Gamma \vdash \mathsf{Impl}(v, \{?p_1 \mapsto v_1, \ldots, ?p_n \mapsto v_n\}) : C^r\tau$ and $\Gamma \vdash f : C^r\tau_1 \to \tau_2$. The reduced expression has a type $C^s\tau_2$:

$$\frac{\dfrac{\Gamma \vdash f : C^r\tau_1 \to \tau_2 \qquad \Gamma \vdash \mathsf{Impl}(v, \{?p_i \mapsto v_i \mid p_i \in r\}) : C^r\tau_1}{\Gamma \vdash f\ (\mathsf{Impl}(v, \{?p_i \mapsto v_i \mid p_i \in r\})) : \tau_2}}{\Gamma \vdash \mathsf{Impl}(f\ (\mathsf{Impl}(v, \{?p_i \mapsto v_i \mid p_i \in r\})), \{?p_i \mapsto v_i \mid p_i \in s\}) : C^s\tau_2}$$

Case (*lookup*): $e = \mathsf{lookup}_{?p_i}(\mathsf{Impl}(v, \{\ \ldots, ?p_i \mapsto v_i \ldots\}))$. The last rule in the type derivation must have been (*lookup*) with $\tau = \mathsf{num}$ and an assumption $\Gamma \vdash \mathsf{Impl}(v, \{\ \ldots, ?p_i \mapsto v_i \ldots\}) : C^{\{\ldots, ?p_i, \ldots\}}\tau$, which requires $\Gamma \vdash v_i : \mathsf{num}$.

Case (*merge, split, letimpl*): Similar. In all three cases, the last typing rule in the derivation of e guarantees that all values of all implicit parameters that are required for the reduction are available. □

Theorem 33 (Progress). *If* $\vdash e : \tau$ *then either* e *is a value or there exists* e' *such that* $e \rightsquigarrow e'$

Proof. By rule induction over $\vdash$.

Case (*num,abs,var,app,proj,tup*): As before, using the adapted canonical forms lemma (Lemma 30) for (*app*) and (*proj*).

Case (*counit*): $e = \mathsf{counit}_{\mathsf{use}}\ e_1$. If e_1 is not a value, it can be reduced using (*ctx*) with context $\mathsf{counit}_{\mathsf{use}}\ \_$, otherwise it is a value. From Lemma 24, $e_1 = \mathsf{Impl}(v, \{\,\})$ and so we can apply (*counit*) reduction rule.

Case (*cobind*): $e = \mathsf{cobind}_{r,s}\ e_1\ e_2$. If e_1 is not a value, reduce using (*ctx*) with context $\mathsf{cobind}_{r,s}\ \_\ e$. If e_2 is not a value reduce using (*ctx*) with context $\mathsf{cobind}_{r,s}\ v\ \_$. If both are values, then from Lemma 30, we have $e_2 = \mathsf{Impl}(v, \{?p_i \mapsto v_i \mid ?p_i \in r \cup s\})$ and we apply the (*cobind*) reduction.

Case (*merge*): $e = \mathsf{merge}_{r,s} e_1$. If e_1 is not a value, reduce using (*ctx*) with context $e = \mathsf{merge}_{r,s}\ \_$. If e_1 is a value, it must be a pair of values $(\mathsf{Impl}(v, \{?p_i \mapsto v_i \mid ?p_i \in r\}), \mathsf{Impl}(v', \{?p_i \mapsto v_i \mid ?p_i \in s\}))$ using Lemma 24 and it can reduce using (*merge*) reduction.

Case (*impl*): $e = \mathsf{Impl}(e', \{?p_1 \mapsto e_1, \ldots, ?p_n \mapsto e_n\})$. If e is not a value, reduce using (*ctx*) with context $\mathsf{Impl}(\_, \{?p_1 \mapsto e_1, \ldots, ?p_n \mapsto e_n\})$. If e_i is not a value, reduce using (*ctx*) with context $\mathsf{Impl}(v, \{?p_1 \mapsto v_1, \ldots, ?p_{i-1} \mapsto v_{i-1}, ?p_i \mapsto \_, ?p_{i+1} \mapsto v_{i+1}, \ldots ?p_n \mapsto e_n\})$. Otherwise, $e, e_0, \ldots, e_n$ are values and so $\mathsf{Impl}(e', \{?p_1 \mapsto e_1, \ldots, ?p_n \mapsto e_n\})$ is also a value.

Case (*split*, *letimpl*): Similar. Either sub-expression is not a value, or the type guarantees that it is a comonadic value with implicit parameter bindings that enable the (*split*) or (*letimpl*) reduction, respectively. □

Theorem 34 (Safety of context-aware language with implicit parameters)**.** *If* $\Gamma \vdash e : \tau$ *and* $e \rightsquigarrow^{*} e'$ *then either* e' *is a value of type* τ *or there exists* e'' *such that* $e' \rightsquigarrow e''$ *and* $\Gamma \vdash e'' : \tau$.

Proof. Rule induction over $\rightsquigarrow^{*}$ using Theorem 32 and Theorem 33. □

5.5 GENERALIZED SAFETY OF COMONADIC EMBEDDING

In Section 5.4.1 and Section 5.4.2, we proved the safety property of two concrete context-aware programming languages based on the coeffect language framework. The proofs for the two systems were very similar and relied on the same key principle.

The principle is that the coeffect annotation r on the type modelling the indexed comonad structure $C^r\tau$ in the target language guarantees that the comonadic value will provide the necessary context. As a result the reductions for operations accessing the context do not get stuck. In case of dataflow, prev_n can always access the tail of the stream and and $\mathsf{counit}_{\mathsf{use}}$ can always access the head (because the stream has a sufficient number of elements). In case of implicit parameters, the context passed to $\mathsf{lookup}_{?p}$ will always contain a binding for $?p$.

Our core functional target language is not expressive enough to capture the relationship between the coeffect annotation and the structure of the Df or Impl value and so we resorted to adding those as ad-hoc extensions. However, given a target language with a sufficiently expressive type system, the properties proved in Section 5.4 would be guaranteed directly by the target language. This includes dependently-typed languages such as Idris or Agda [13, 12], but type-level numerals and sets can also be encoded in the Haskell type system [79].

In other words, the flat coeffect type system, together with the translation for introduced in this chapter, can be embedded into a Haskell-like languages and it can provide a succinct and safe way of implementing context-aware domain specific languages.

COEFFECTS FOR LIVENESS. As an example, we consider the third instance of coeffect calculus that was discussed in Chapter 4. If we wanted to follow the development in the previous section for liveness, we would extend the target language with two kinds of expressions, Dead representing a dead context with no value and Live representing a context with a value:

$$e \quad = \quad \ldots \mid \mathsf{Dead} \mid \mathsf{Live}\ e$$

The typing rules promote the information about whether a value is available into the type-level and so a context carrying a live value is marked as $C^{\mathsf{L}}\tau$ while a dead context has a type $C^{\mathsf{D}}\tau$.

$$(\textit{live})\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{Live}\ e : C^{\mathsf{L}}\tau} \qquad (\textit{dead})\ \frac{}{\Gamma \vdash \mathsf{Dead} : C^{\mathsf{D}}\tau}$$

Finally, we need to add reduction rules that define the meaning of the comonadically-inspired operations for liveness. Those follow the definitions given in Example 10 when discussing the categorical semantics:

$$\begin{aligned}
&(\textit{counit}) && \mathsf{counit}_{\mathsf{L}}\ (\mathsf{Just}\ v) \rightsquigarrow v \\
&(\textit{cobind-1}) && \mathsf{cobind}_{\mathsf{L},\mathsf{L}}\ f\ (\mathsf{Live}\ v) \rightsquigarrow \mathsf{Live}\ (f\ v) \\
&(\textit{cobind-2}) && \mathsf{cobind}_{\mathsf{D},\mathsf{L}}\ f\ (\mathsf{Live}\ v) \rightsquigarrow \mathsf{Live}\ (f\ \mathsf{Dead}) \\
&(\textit{cobind-3}) && \mathsf{cobind}_{\mathsf{L},\mathsf{D}}\ f\ v \rightsquigarrow \mathsf{Dead} \\
&(\textit{cobind-4}) && \mathsf{cobind}_{\mathsf{D},\mathsf{D}}\ f\ v \rightsquigarrow \mathsf{Dead}
\end{aligned}$$

This language extension is safe because the reductions respect the typing of the comonadically-inspired operations. The $\mathsf{counit}_{\mathsf{use}}$ reduction does not get stuck for well-typed terms because $\mathsf{use} = \mathsf{L}$ in the coeffect algebra and thus its argument is of type $C^{\mathsf{L}}\tau$ and will always be a value $\mathsf{Live}\ v$.

Similarly, the when reducing $\mathsf{cobind}_{r,s}$, the typing ensures that the value passed as the second argument is of type $C^{r \circledast s}\tau_1$. In case of liveness, $r \circledast s = \mathsf{L}$ if either $r = \mathsf{L}$ or $s = \mathsf{L}$. This means that reductions (*cobind-1*) and (*cobind-2*) will not get stuck because the value will be $\mathsf{Live}\ v$ and not Dead. The reduction rules also preserve typing – the resulting value is of type $C^{s}\tau_2$, that is $\mathsf{Live}\ v$ for (*cobind-1*), (*cobind-2*) and Dead for (*cobind-3*) and (*cobind-4*).

ENCODING LIVENESS IN HASKELL. The liveness example can be encoded in Haskell using type-level features such as generalized algebraic data types (GADTs) and type families [63, 129, 19], which encode some of the features known from dependently-typed languages such as Agda [12]. We do not aim to give a complete implementation, but to show that such encoding is possible and would provide the necessary safety guarantees.

We first define types D and L to capture the coeffect annotations. Then we define a comonadically-inspired type C r a as a GADT with cases for Live and Dead contexts. The type parameter r represents a coeffect annotation:

```
data L
data D

data C r a where
  Live  :: a → C L a
  Dead :: C D a
```

The definition matches with the typing rules (*live*) and (*dead*). The coeffect annotation for a live value is L and the annotation for a dead value is D. To give the type of cobind, we need a type-level function that encodes the operations of the flat coeffect algebra. We model $\circledast$ as Seq a b. The operation

is defined on types L and D and returns a type D if and only if both its arguments are D:

```
type family Seq r s :: *
  type instance Seq D D = D
  type instance Seq L s = L
  type instance Seq r L = L
```

The counit and cobind operations can then be defined as Haskell functions that have types corresponding to the typing rules (*counit*) and (*cobind*) given in Figure 29:

```
counit  :: C L a → a
cobind  :: (C r a → b) → C (Seq r s) a → C s b
```

Here, the additional type parameter is used as a phantom type [59] and ensures that counit can only be called on a context that contains a value and so calling the operation is not going to fail. Similarly, the type of the cobind operation now guarantees that if a function (used as the first argument) or the result require a live context, it will be called with a value C L a that is guaranteed to contain a value.

COEFFECTS IN DEPENDENTLY-TYPED LANGUAGES. If we use the above encoding, type preservation is guaranteed by the type system of the target language. The equivalent of the progress property is guaranteed by the fact that the the implementation of the operations is well-defined.

It is worth noting that this is where coeffects need a target language with a more expressive type system than monads. For monadic computations, it is sufficient to use a type M a which represents that *some* effect may happen. The the type does not specify which effects and, indeed, this means that *all possible effects* may happen.

With coeffects, we need to use indexed comonads C r a where the annotation r specifies what context may be required. Without the annotation, a type C a would represent a comonadic context that has *all possible context* available, which is rarely useful in practice.

5.6 RELATED CATEGORICAL STRUCTURES

Related work leading to coeffects has already been discussed in Chapter 2 and we covered work related to individual concepts throughout the thesis. However, there is a number of related categorical structures that are related to our *indexed comonads* (Section 5.2.4) that deserve additional discussion.

In Section 5.6.1, we discuss related approaches to adding indices to categorical structures (mostly monads). In Section 5.6.2, we discuss a question that often arises when discussing coeffects and that is *when is a coeffect (not) an effect?*

5.6.1 *Indexed categorical structures*

Ordinary comonads have the *shape preservation* property [78]. Intuitively, this means that the core comonad structure does not provide a way of modeling computations where the additional context changes 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 comonad, in the computation $\mathsf{cobind}_{r,s}\, f$, the shape of the context changes from providing implicit parameters $r \cup s$ to providing just implicit parameters s. Thus *indexed comonads* are a generalization of *comonads* that captures structures that fail to form a comonad without indexing. In the rest of the section, we look at work that discusses indexing in the context of *monads.*

FAMILIES OF MONADS. When linking effect systems and monads, Wadler and Thiemann [67] propose a *family of monads* as the categorical structure. The dual structure, *family of comonads*, is defined as follows.

Definition 10. *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.*

A *family of comonads* is not as expressive as an *indexed comonads*. Many indexed comonads cannot be captured by a family of comonads. This is because each of the data types needs to form a comonad separately. For example, our indexed Maybe 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 $\iota_{r\circledast s,s}$.

PARAMETRIC EFFECT MONADS. Parametric effect monads introduced by Katsumata [52] (independently to our indexed comonads) are closely related to our definition. Although presented in a more general categorical framework (and using monads), the model (i) defines the unit operation only on the unit of a monoid and (ii) the bind operation composes effect annotations using the provided monoidal structure.

5.6.2 *When is coeffect not a monad*

Coeffect systems differ from effect systems in three important ways:

- Semantically, coeffects capture 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 dataflow). In terms of program analysis [56], monads capture forward dataflow analyses and comonads correspond to backward dataflow analyses.
- Syntactically, coeffect calculi use a richer algebraic structure with pointwise composition, sequential composition and context merging ($\oplus$, $\circledast$, and $\wedge$) while most effect systems only use a single operation for sequential composition (used by monadic bind). Effect systems may use a richer algebraic structure to support additional language constructs such as conditionals [71, 91], but not for abstraction and application.
- Syntactically, the second difference is in the lambda abstraction (*abs*). In coeffect systems, the context demands of the body can be split between (or duplicated at) declaration site and call site, while lambda abstraction in monadic effect systems always defer all effects – creating a function value has no effect.

Despite the differences, our implicit parameters resemble, in many ways, the *reader* monad. As discussed in Section 5.6.3, the *reader* monad is semantically equivalent to the *product* comonad when we consider just sequential composition. For a language with lambda abstraction, we need a slight extension to the usual treatment of monads in order to model implicit parameters using a monad.

5.6.3 *When is coeffect a monad*

Implicit parameters can be captured by a monad, but *just* a monad is not enough. Lambda abstraction in effect systems does not provide a way of splitting the context demands between declaration site and call site (or, semantically, combining the implicit parameters available in the scope where the function is defined 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 $r \to \mathsf{num}$ is a lookup function for reading implicit parameter values that is defined on a set r. The two definitions are:

$$C^r\tau = \tau \times (r \to \sigma) \qquad \textit{(product comonad)}$$
$$M^r\tau = (r \to \sigma) \to \tau \qquad \textit{(reader monad)}$$

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 [76], when used to model computation semantics, the two representations are equivalent:

Remark 35. *Computations modelled as $C^r\tau_1 \to \tau_2$ using the product comonad are isomorphic to computations modelled as $\tau_1 \to M^r\tau_2$ using the reader monad via currying/uncurrying isomorphism.*

Proof. The isomorphism is demonstrated by the following equation:

$$\begin{aligned} C^r\tau_1 \to \tau_2 &= (\tau_1 \times (r \to \sigma)) \to \tau_2 \\ &= \tau_1 \to ((r \to \sigma) \to \tau_2) = \tau_1 \to M^r\tau_2 \end{aligned}$$

This equivalence shows an intriguing relationship between the *product comonad* and *reader monad*, but it cannot be extended beyond that. In particular, comonads that model dataflow computations or liveness do not have a corresponding monadic structure. This 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 $\mathsf{merge}_{r,s}$ operation to model context merging. This can be supported in monadic computations by adding an additional operation discussed next.

DELAYING EFFECTS IN MONADS. In the syntax of the language, the above difference is manifested by the (*abs*) rules for monadic effect systems and comonadic coeffect systems. The following listing shows the two rules side-by-side, using the effect system notation for both of them:

$$(\textit{cabs})\ \frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2\ \&\ r \cup s}{\Gamma \vdash \lambda x.e : \tau_1 \xrightarrow{s} \tau_2\ \&\ r} \qquad (\textit{mabs})\ \frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2\ \&\ r \cup s}{\Gamma \vdash \lambda x.e : \tau_1 \xrightarrow{r \cup s} \tau_2\ \&\ \emptyset}$$

In the comonadic (*cabs*) rule, the implicit parameters of the body are split. However, the monadic rule (*mabs*) places all demands 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 performs 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 demands $r \cup s$ are split – and thus guarantee determinism of the function at run-time.

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 $M^{r}\tau$:

$$\mathsf{delay}_{r,s} \;:\; (\tau_1 \to (r \cup s \to \sigma) \to \tau_2) \to ((r \to \sigma) \to \tau_1 \to (s \to \sigma) \to \tau_2)$$

This suggests that the *reader monad* is a special case among monads. Our work suggests that passing read-only information to a computation is better captured by a product comonad, which also matches the intuition – read-only information is a *contextual capability*.

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 demands in lambda abstraction. Can we also go the other way round and *restrict* the comonadic semantics so that all demands are delayed as in the (*mabs*) rule, thus modelling fully dynamically scoped parameters?

This is, indeed, 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 an indexed 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:

$$\begin{aligned}
&\mathsf{merge}_{r,s} \;:\; C^{r}\alpha \times C^{s}\beta \to C^{r\cup s}(\alpha \times \beta) && (\textit{normal})\\
&\mathsf{merge}_{r,s} \;:\; C^{\emptyset}\alpha \times C^{s}\beta \to C^{s}(\alpha \times \beta) && (\textit{restricted})
\end{aligned}$$

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. The ability to restrict operations to partial functions has been used in the semantics of effectful computations by Tate [107].

5.7 SUMMARY

In the previous chapter, we defined a *type system* for flat coeffect calculi that uniformly captures the shared structure of context-aware computations. In this chapter, we completed the unification by providing *semantics* for flat coeffect calculi and proving the *safety* of coeffect languages for dataflow and implicit parameters. The semantics shown here also guides the implementation that is discussed later in Chapter 7.

The development presented in this chapter follows the well-known example of effects and monads. We introduced the notion of *indexed comonad*, which generalizes comonads and adds additional operations needed to provide categorical semantics of the flat coeffect calculus and we demonstrated how implicit parameters, liveness and dataflow computations form indexed comonads.

We then used the comonadic semantics to define a *comonadically-inspired translation* that turns programs written in a domain-specific coeffect language into a functional target language. This is akin to the Haskell "do" notation for monads. Finally, we extended the target language with concrete implementations of comonadic operations for dataflow and implicit parameters and we presented a syntactic safety proof. In summary, the proof states that well-typed context-aware programs written in a coeffect language *do not go wrong* (when translated to a simple functional language and evaluated).

The proof relies on the fact that coeffect annotations (provided by the coeffect type system) guarantee that the required context is available in the comonadic value that represents the context and we also discussed how this would guarantee safety in languages with sufficiently expressive type system such as Haskell.

In the following chapter, we move from *flat* coeffect calculi, tracking whole-context properties to *structural* coeffect calculi, tracking per-variable information, thus covering systems from the second half of Chapter 3.

6 THE STRUCTURAL COEFFECT CALCULUS

In Chapter 3, we discussed two notions of context. Context-aware programming languages that capture whole-context properties were generalized by the *flat coeffect calculus* in Chapters 4 and 5. Here, we consider per-variable contextual properties and we introduce the *structural coeffect calculus*.

The flat coeffect system captures a number of interesting use-cases. For some of those (liveness and dataflow), flat coeffects provide only imprecise approximation (for example, marking the whole context as live rather than marking individual variables). Dataflow and liveness (but not implicit parameters) can be also seen as per-variable properties. For those, structural coeffect systems capture more precise information about the context. However, we also look at other applications that arise from the work on substructural logics discussed in Section 2.4.

We mirror the development for flat coeffect calculus and develop a small calculus with a type system that captures per-variable contextual properties. We outline its categorical semantics and use it as a basis for a translation that turns well-typed programs in context-aware languages into well-typed programs in a simple target functional language. We prove syntactic safety for a sample target language, showing that "well-typed context-aware programs do not go wrong".

CHAPTER STRUCTURE AND CONTRIBUTIONS

- We present a *structural coeffect calculus* as a type system that is parameterized by a *structural coeffect algebra* (Section 6.2). We show how the system captures pre-variable liveness and dataflow information, as well as a calculus for bounded reuse (checking how many times is a variable accessed).
- We present a syntax-directed version of the calculus that is used to obtain unique typing derivation for programs in structural coeffect calculus (Section 6.3). Unlike in flat systems, the procedure for choosing a unique typing derivation is common to all structural systems.
- We discuss the equational theory of the calculus. We show that type-preservation holds for all examples of the structural calculus we consider, for both call-by-name and call-by-value reductions (Section 6.5) and we explore a number of extensions to the minimal calculus based on λ-calculus, including subcoeffecting and let binding (Section 6.4).
- We extend the indexed comonads introduced in the previous chapter to *structural indexed comonads* and use them to provide the semantics of structural coeffect calculus (Section 6.6). As with the flat version, the theory serves as a motivation for syntactic translational semantics.
- We give a *translational semantics* (Section 6.7) that translates programs from the structural coeffect calculus into a simple functional language with uninterpreted comonadically-inspired primitives. We give concrete operational semantics for the target language for one of our sample languages and show that well-typed programs, produced by translation from the coeffect calculus do not go wrong.

6.1 INTRODUCTION

Compared to Chapter 4, the structural coeffect calculi we consider are more homogeneous and so finding the common pattern is easier. However, the systems are more complicated as they need to keep annotations attached to individual variables and thus require explicit structural rules. Before looking at the system, we briefly consider the most important related work.

6.1.1 *Related work*

In the previous chapter, we discussed the correspondence between coeffects and effects (and between comonads and monads). As noted in Section 3.1.3, the λ-calculus is asymmetric in that an expression has multiple inputs (variables in the context), but just a single result (the resulting value). So, while there is only one notion of monadic effect system, there are two separate notions of coeffect system - one that keeps coeffect annotations per-environment and one that keeps coeffect annotations per-variable.

The work in this chapter is related to substructural type systems [126]. Substructural systems remove some or all of *weakening*, *contraction* and *exchange* rules. In contrast, our systems keep all three structural rules, but use them to manipulate the coeffect annotations in a way that matches the variable manipulations.

Our work follows the "language semantics" style in that we provide semantics to the terms of ordinary λ-calculus. By contrast the closely related work on Contextual Modal Type Theory (CMTT) [70] follows the meta-language style. CMTT extends the terms and types of a language with constructs for explicitly manipulating the context. Variables of type $A[\Psi]$ denote 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 (indexed comonads only appear in the semantics).

Our 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 $x : \tau_1, y : \tau_2 \mathbin{@} \langle s, t \rangle$. A benefit of this approach is that the typing judgements have the same structure as those of the flat coeffect calculus. As discussed in Section 8.1, this makes it possible to unify the two systems.

6.2 STRUCTURAL COEFFECT CALCULUS

In the structural coeffect calculus, functions are annotated with a primitive (scalar) coeffect annotations. A vector of variables forming the free-variable context is annotated with a vector of coeffect annotations. These annotations differ for various coeffect calculi; their properties are captured by the definition of *structural coeffect scalar* below. The scalar annotations can be e. g. integers (how many past values we need) or values of a two-point lattice specifying whether a variable is live or not. The expressions and types of the structural coeffect calculus are defined as follows:

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

The expressions and types of structural coeffect calculus are similar to those of the flat coeffect calculus with two differences. First, we omit the `let` construct in the core language. In structural coeffects, let binding can be defined

as a derived rule using abstraction and application (Section 6.4.1). Second, the coeffect annotations r, s, t on function type now range over values of a *structural coeffect scalar*.

6.2.1 *Structural coeffect algebra*

The *structural coeffect scalar* structure is similar to that of *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 pointwise composition, as well as the $\leqslant$ relation.

Definition 11. *A structural coeffect scalar $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leqslant)$ is a set $\mathcal{C}$ together with elements $\mathsf{use}, \mathsf{ign} \in \mathcal{C}$, binary operations $\circledast, \oplus$ such that $(\mathcal{C}, \circledast, \mathsf{use})$ and $(\mathcal{C}, \oplus, \mathsf{ign})$ are monoids and a binary relation $\leqslant$ such that $(\mathcal{C}, \leqslant)$ is a pre-order. That is, for all $r, s, t \in \mathcal{C}$:*

$$r \circledast (s \circledast t) = (r \circledast s) \circledast t \qquad \mathsf{use} \circledast r = r = r \circledast \mathsf{use} \qquad \text{(monoid)}$$

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

$$\text{if } r \leqslant s \text{ and } s \leqslant t \text{ then } r \leqslant t \qquad t \leqslant t \qquad \text{(pre-order)}$$

In addition, the following distributivity axioms hold:

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

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

The structural coeffect scalar structure resembles flat coeffect algebra, but it differs in two important ways:

- The $\oplus$ operation of structural coeffect scalar is not required to be idempotent. In structural systems, we can track individual variable accesses and not requiring idempotence allows interesting systems such as that for bounded reuse (Section 6.6.5).
- 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 demands. Structural systems *append vectors* of annotations instead of *merging annotations* and so $\wedge$ is no longer needed.

In the structural coeffect calculus, the scalar coeffect structure is supplemented by a vector structure. The vector structure is used to manipulate vectors of coeffect scalars attached to a variable context. The required structure is captured by the following definition.

Definition 12. *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 $\boldsymbol{r}, \boldsymbol{s}, \boldsymbol{t}$, ranging over structural coeffect scalars indexed by vector lengths $m, n \in \mathbb{N}$.*
- *A family of operations (indexed by the vector length) that construct a vector from scalars $\langle - \rangle_n : \mathcal{C} \times \ldots \times \mathcal{C} \to \mathcal{C}^n$ and an operation that returns the vector length such that $\mathit{len}(\boldsymbol{r}) = n$ for $\boldsymbol{r} : \mathcal{C}^n$*
- *A pointwise extension of the $\circledast$ operator written as $t \circledast \boldsymbol{s}$ 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 $\#_{n,m} : \mathcal{C}^n \times \mathcal{C}^m \to \mathcal{C}^{n+m}$ that is bijective and is used in both directions – for vector concatenation and for splitting – which is defined as $\langle r_1, \ldots, r_n \rangle \#_{n,m} \langle s_1, \ldots, s_m \rangle = \langle r_1, \ldots, r_n, s_1, \ldots, s_m \rangle$*

a.) Syntax-driven typing rules:

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

$$(const)\ \frac{}{() \mathbin{@} \langle\rangle \vdash n : \mathsf{num}}$$

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

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

b.) Structural rules for context manipulation:

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

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

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

Figure 33: Type system for the structural coeffect calculus

The fact that the tensor product $+\!\!+_{n,m}$ is indexed by the lengths of the two vectors means that we can use it unambiguously for concatenating two vectors and splitting of a vector, provided that the lengths of the resulting vectors are known. In the following text, we usually omit the indices and write just $\mathbf{r}+\!\!+\mathbf{s}$, 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 coeffect annotations as *containers* [2]. This approach is used in Section 8.1 to unify flat and structural systems.

6.2.2 *Structural coeffect types*

The type system for the structural coeffect calculus is shown in Figure 33. It is similar to substructural type systems [126] in how it handles free variable contexts. In the type system for flat coeffects (Section 4.2.2), the (*var*) rule implicitly allows *weakening* and *exchange* by ignoring other variables in the context and (*app*) implicitly allows *contraction* by passing the same context to both sub-expressions.

In the structural system, this is made explicit by adding *structural rules*. While substructural type systems usually remove some of the rules, we keep all three and use them to track how variables are used. This is done by manipulating the coeffect annotations in parallel with manipulating the variable contexts. As in substructural type systems, (*app*) checks the types of sub-expressions in disjoint parts of the free variable contexts and (*var*) requires the context to contain exactly one variable. The typing rule for let binding in structural coeffects is a derived rule and we discuss it later in Section 6.4.1.

VARIABLE CONTEXTS. In Chapter 4, the free variable context Γ was treated as a set. In the type system for the structural coeffect calculus, the variable context is treated as a vector, with an additional condition that a cannot appear multiple times. We also write $len(-)$ for the length of the vector:

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

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. The notation Γ_1, Γ_2 is used for concatenation of vectors of variables. That is, given a context $\Gamma_1 = \langle x_1 : \tau_1, \ldots, x_n : \tau_n \rangle$ and a context $\Gamma_2 = \langle x_{n+1} : \tau_{n+1}, \ldots, x_m : \tau_m \rangle$ then $\Gamma_1, \Gamma_2 = \langle x_1 : \tau_1, \ldots, x_m : \tau_m \rangle$.

In the typing rules, free variable contexts are annotated with vectors of structural coeffect scalars, written as $x_1 : \tau_1, , x_n : \tau_n \mathbin{@} \langle r_1, \ldots, r_n \rangle$. Meta-variables ranging over coeffect 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 $len(\mathbf{r})$.

SYNTAX-DRIVEN RULES. The syntax-driven rules of the type system are shown in Figure 33 (a). The variable access rule (*var*) annotates the corresponding variable as being accessed using use. As in substructural 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 form of typing rule obviates the ambiguity in splitting of context demands present in the 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. These are shown in Figure 33 (b). The three structural rules are not syntax-directed and allow different transformation of the free variable context. They correspond to the transformations known as *weakening*, *exchange* and *contraction* from substructural systems.

Rule (*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 be *unique*. The (*contr*) rule allows re-using a variable. We can type check sub-expressions using two separate variables 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 demands are combined (semantically, this is where the available context is shared).

6.2.3 *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 6.5, has a more desirable equational theory. By contrast, the flat system allows certain interesting use cases that *rely* on the flexibility of $\wedge$ in lambda abstraction (such as implicit parameters), that cannot be expressed in the structural system.

In flat systems, lambda abstraction splits context demands using $\wedge$ and application combines them using $\oplus$. In the structural version, both of these are replaced with $\#$. The $\wedge$ operation is not needed, but note the use of $\oplus$ in the (*contr*) rule.

This suggests that $\wedge$ and $\oplus$ serve two roles in flat coeffects. First, they are used as over-approximations and under-approximations of $\#$. 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 two values representing the available context, potentially discarding parts of it (under-approximation), while flat application splits the available context (a single value), potentially duplicating parts of it (over-approximation)[1].

Secondly, the operator $\oplus$ is used when the semantics passes a given context to multiple sub-expressions. In flat systems, the context is shared in (*app*) and the additional (*pair*) rule for constructing tuple values, because the sub-expressions may share variables. In structural systems, the sharing is isolated into an explicit contraction rule.

6.2.4 *Examples of structural coeffects*

The structural coeffect calculus above can be instantiated to obtain the 3 structural coeffect calculi presented in Section 3.3. Two of them – structural dataflow and structural liveness provide a more precise tracking of properties that can be tracked using flat systems. Formally, any flat coeffect algebra can be turned into a structural coeffect scalar (by dropping the $\wedge$ operator). This is useful for liveness and dataflow, but it does not yield a practically useful system for the flat algebra coeffect for implicit parameters.

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 scalar. This is the case, for example, for the system tracking bounded variable use (Example 15).

Example 13 (Structural liveness)**.** *The structural coeffect scalar 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 a meet* $\sqcap$*.*

Example 14 (Structural dataflow)**.** *In dataflow, context is annotated with natural numbers and the structural coeffect scalar is formed by* $(\mathbb{N}, +, max, 0, 0, \leqslant)$*.*

These two examples have both flat and structural versions. For them, obtaining the structural 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 dataflow* example from Section 4.2.4.

1 Because of this duality, earlier version of coeffect systems [83] used $\wedge$ and $\vee$.

It is important to note that this gives us a systems with *different* properties. Information is now tracked per-variable rather than for entire contexts. For dataflow, we also need to adapt the typing rule for the `prev` construct. Here, we write $+$ for a pointwise extension of the $+$ operator, such that $\langle r_1, \ldots, r_n \rangle + k = \langle r_1 + k, \ldots, r_n + k \rangle$.

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

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 actually used in the expression e, whereas in the flat coeffect system the rule incremented the annotation for the whole context. Thanks to the structural nature of the system, annotations of variables that do not appear in the expression e can be left unchanged.

Before looking at properties of structural coeffect systems, we consider a system for tracking 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 dataflow, the calculus for bounded variable reuse annotates each variable with an integer. However, the integer now 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.

6.3 CHOOSING A UNIQUE TYPING

In the structural coeffect calculus, the lambda abstraction rule does not introduce ambiguity in the typing. This is in contrast with flat coeffect systems (most importantly, the one for implicit parameters), where lambda abstraction allowed arbitrary splitting of context demands. In structural coeffect systems, the context demands placed on the call site (attached to the function type) are those of the bound variable.

However, the type system for structural coeffect calculus in Figure 33 introduces another kind of ambiguity due to the fact that non-syntax-directed structural rules can be applied repeatedly and in arbitrary order. As with the semantics for flat coeffect calculus in Chapter 5, we define the semantics of the structural coeffect calculus relative to a *typing derivation* and so the meaning of a program depends on the typing derivation chosen. In this section, we specify how to choose the desired *unique* typing derivation, following the example of flat coeffect calculi as discussed in Section 4.3.

6.3.1 *Syntax-directed type system*

In order to choose a unique typing derivation, we follow the example of substructural type systems [126] and introduce a syntax-directed version of the type system. This replaces the non-syntax-directed rules for weakening, exchange and contraction with more complexity in the rules where contexts are combined (*app*) and variables removed (*abs*).

Given a typing derivation in the deterministic syntax-directed type system, we then choose a typing derivation in the original type system that uniquely specifies how to apply weakening, contraction and exchange. The

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

$$(const)\ \frac{}{() \mathbin{@} \langle\rangle \vdash n : \mathsf{num}}$$

$$(app)\ \frac{\Gamma_1 \mathbin{@} \mathbf{r} \vdash e_1 : \tau_1 \xrightarrow{t} \tau_2 \quad \Gamma_2 \mathbin{@} \mathbf{s} \vdash e_2 : \tau_1}{\Gamma \mathbin{@} \mathbf{c} \vdash e_1\ e_2 : \tau_2} \quad \Gamma \mathbin{@} \mathbf{c} = \mathsf{mergevars}(t, \Gamma_1 \mathbin{@} \mathbf{r}, \Gamma_2 \mathbin{@} \mathbf{s})$$

$$(abs)\ \frac{\Gamma_1 \mathbin{@} \mathbf{t} \vdash e : \tau_2}{\Gamma_2 \mathbin{@} \mathbf{r} \vdash \lambda x : \tau_1 . e : \tau_1 \xrightarrow{s} \tau_2} \quad (\Gamma_2 \mathbin{@} \mathbf{r}), s = \mathsf{findvar}_{x,\tau_1}(\Gamma_1 \mathbin{@} \mathbf{t})$$

$$\mathsf{findvar}_{x,\tau}(\Gamma \mathbin{@} \mathbf{t}) = (\Gamma_1, \Gamma_2 \mathbin{@} \mathbf{t}_1 \mathbin{+\!\!+} \mathbf{t}_2), s \quad \text{where}$$
$$len(\Gamma_1) = len(\mathbf{t}_1) \text{ and } len(\Gamma_2) = len(\mathbf{t}_2)$$
$$x{:}\tau \in \Gamma \text{ and } \Gamma \mathbin{@} \mathbf{t} = \Gamma_1, x{:}\tau, \Gamma_2 \mathbin{@} \mathbf{t}_1 \mathbin{+\!\!+} \langle s \rangle \mathbin{+\!\!+} \mathbf{t}_2$$

$$\mathsf{findvar}_{x,\tau}(\Gamma \mathbin{@} \mathbf{t}) = (\Gamma \mathbin{@} \mathbf{t}), \mathsf{ign} \qquad (\text{otherwise})$$

$$\mathsf{mergevars}(t, \Gamma_1 \mathbin{@} \mathbf{r}, \Gamma_2 \mathbin{@} \mathbf{s}) = \Gamma_1', \Gamma_2', \Gamma \mathbin{@} \mathbf{r}' \mathbin{+\!\!+} (t \circledast \mathbf{s}') \mathbin{+\!\!+} \mathbf{c} \quad \text{where}$$
$$\Gamma_1 \mathbin{@} \mathbf{r} = x_1{:}\tau_1, \ldots, x_n{:}\tau_n \mathbin{@} \langle r_1, \ldots, r_n \rangle$$
$$\Gamma_2 \mathbin{@} \mathbf{s} = y_1{:}\tau_1', \ldots, y_m{:}\tau_n' \mathbin{@} \langle s_1, \ldots, s_m \rangle$$
$$\Gamma \mathbin{@} \mathbf{c} = z_1{:}\tau_1'', \ldots, z_k{:}\tau_k'' \mathbin{@} \langle c_1, \ldots, c_k \rangle$$
$$\text{such that } \forall l \in \{1 \ldots k\}\ \exists i, j.\ (z_l{:}\tau_l'' = x_i{:}\tau_i = y_j{:}\tau_j')$$
$$\text{and } c_l = r_i \oplus (t \circledast s_j)$$
$$\Gamma_1 \mathbin{@} \mathbf{r} = x_1{:}\tau_1, \ldots, x_n{:}\tau_n \mathbin{@} \langle r_1, \ldots, r_n \rangle \text{ such that } x_i{:}\tau_i \notin \Gamma$$
$$\Gamma_2 \mathbin{@} \mathbf{s} = y_1{:}\tau_1', \ldots, y_m{:}\tau_n' \mathbin{@} \langle s_1, \ldots, s_m \rangle \text{ such that } y_i{:}\tau_i' \notin \Gamma$$

Figure 34: Syntax-directed type system for the structural coeffect calculus

algorithm given in Proposition 38 inserts only the necessary structural rules to rearrange variable context into the shape required by the assumptions.

The syntax-directed version of the type system is shown in Figure 34. The typing rules for variables (*var*) and constants (*const*) are the same as before. The two interesting rules are lambda abstraction and application.

LAMBDA ABSTRACTION. In the lambda abstraction (*abs*) rule in Figure 33, we assume that the bound variable is the last variable of the context. In the syntax-directed system, we do not make the same assumption. Instead, we use an auxiliary function $\mathsf{findvar}_{x,\tau}$ that takes a typing context $\Gamma \mathbin{@} \mathbf{t}$ and returns a context with the variable x removed together with the coeffect originally attached to the variable. The $\mathsf{findvar}_x$ function is defined by two disjoint cases. The case when the variable x is not present in the context corresponds to the (*weak*) structural rule.

FUNCTION APPLICATION. The (*app*) rule in Figure 33 assumes that the variable contexts of the two sub-expressions can be merged. This requires that they contain disjoint variables, which can be always obtained by exchange and contraction. In the syntax-driven system, we merge coeffects of shared variables explicitly. This is done in the mergevars function.

As with $\mathsf{findvar}_{x,\tau}$, the mergevars function is fully deterministic. It returns context consisting of three parts. Parts Γ_1 and Γ_2 represent variables that appear only in the first or the second context; part Γ contains common variables. The coeffect annotations corresponding to Γ_1 are the original annotations from $\mathbf{r}$; the coeffects corresponding to Γ_2 are composed with the coeffect of the function value $t \circledast \mathbf{s}'$ as in the original (*app*) rule. Finally, for shared variables, the coeffect is obtained by point-wise composition (as in contraction) of the coeffect for the two contexts $r_i \oplus (t \circledast s_j)$. The first coeffect corresponds to the context demands in the sub-expression e_1 and the second coeffect corresponds to the function argument e_2 sequentially composed with the coeffect t of the function (as in ordinary application rule).

6.3.2 *Properties*

The syntax-directed type checking presented in the previous section gives a unique typing derivation that can be automatically turned into one of the valid typing derivations of the original type system presented in Figure 33. This gives us a unique typing derivation for the structural coeffect calculus. As with the unique typing derivation for the flat coeffect system, the chosen typing derivation is used to give semantics of terms of the structural coeffect calculus. We also note that a well-typed program in the original type system has a typing derivation in the syntax-driven version.

As when discussing uniqueness of typing for flat coeffect systems (Section 4.3), we first give an inversion lemma (Lemma 36) and then prove uniqueness of typing (Theorem 37).

Lemma 36 (Inversion lemma for syntax-directed structural coeffects)**.** *For the type system defined in Figure 34:*

1. *If* $\Gamma \mathbin{@} \mathbf{c} \vdash x : \tau$ *then* $\Gamma = x{:}\tau$ *and* $\mathbf{c} = \langle \mathsf{use} \rangle$.
2. *If* $\Gamma \mathbin{@} \mathbf{c} \vdash n : \tau$ *then* $\Gamma = ()$ *and* $\tau = \mathsf{num}$ *and* $\mathbf{c} = \langle\rangle$.
3. *If* $\Gamma \mathbin{@} \mathbf{c} \vdash e_1\ e_2 : \tau_2$ *then there is some* $\Gamma_1, \Gamma_2, \tau_1$ *and some* $t, \mathbf{r}, \mathbf{s}$ *such that* $\Gamma_1 \mathbin{@} \mathbf{r} \vdash e_1 : \tau_1 \xrightarrow{t} \tau_2$ *and* $\Gamma_2 \mathbin{@} \mathbf{s} \vdash e_2 : \tau_1$ *and also* $\Gamma \mathbin{@} \mathbf{c} = \mathsf{mergevars}(t, \Gamma_1 \mathbin{@} \mathbf{r}, \Gamma_2 \mathbin{@} \mathbf{s})$.
4. *If* $\Gamma \mathbin{@} \mathbf{c} \vdash \lambda x{:}\tau_1.e : \tau$ *then there is some* Γ', τ_2 *and some* $s, \mathbf{t}$ *such that* $\Gamma' \mathbin{@} \mathbf{t} \vdash e : \tau_2$ *and* $\tau = \tau_1 \xrightarrow{s} \tau_2$ *and also* $(\Gamma \mathbin{@} \mathbf{c}), \tau_1, s = \mathsf{findvar}_x(\Gamma_1 \mathbin{@} \mathbf{t})$.

Proof. Follows from the individual rules given in Figure 34. □

Theorem 37 (Uniqueness of syntax-directed structural coeffects)**.** *In the syntax-directed type system for structural coeffects defined in Figure 34, when* $\Gamma \mathbin{@} \mathbf{r} \vdash e : \tau$ *and* $\Gamma \mathbin{@} \mathbf{r}' \vdash e : \tau'$ *then* $\tau = \tau'$ *and* $\mathbf{r} = \mathbf{r}'$.

Proof. Suppose that (A) $\Gamma \mathbin{@} \mathbf{c} \vdash e : \tau$ and (B) $\Gamma \mathbin{@} \mathbf{c}' \vdash e : \tau'$. We show by induction over the typing derivation of $\Gamma \mathbin{@} \mathbf{c} \vdash e : \tau$ that $\tau = \tau'$ and $\mathbf{c} = \mathbf{c}'$.

Case (*abs*): $e = \lambda x : \tau_1.e_1$. Then $\tau = \tau_1 \xrightarrow{c} \tau_2$ for some τ_2 and $\Gamma' \mathbin{@} \mathbf{t} \vdash e : \tau_2$ for some $\Gamma', \mathbf{t}$ and also $(\Gamma \mathbin{@} \mathbf{c}), \tau_1, s = \mathsf{findvar}_x(\Gamma' \mathbin{@} \mathbf{t})$. By case (4) of Lemma 36, the final rule of the derivation (B) must have also been (*abs*) and this derivation has a sub-derivation with a conclusion $\Gamma \mathbin{@} \mathbf{c}' \vdash e : \tau_2'$. By the induction hypothesis $\tau_2 = \tau_2'$ and $\mathbf{c} = \mathbf{c}'$ and therefore also so $\tau = \tau'$. Although $\mathsf{findvar}_x$ is a relation, it allows only one possible result (because the type of the bound variable matches the type annotation).

Cases (*var*), (*const*) are direct consequence of Lemma 36.

Case (*app*) similarly to (*abs*). □

As noted earlier, unique typing derivations obtained using the syntax-directed type system given in Figure 34 can be automatically turned into typing derivations of the original (non-syntax-directed) structural coeffect type system in Figure 33. Unlike in the flat coeffect system, this does not determine how context demands are split (as this is done deterministically to match the variable bindings), but it specifies how are the structural rules (weakening, exchange and contraction) applied. The following proposition provides the details.

Proposition 38 (Choosing a unique typing derivation)**.** *If* $\Gamma \mathbin{@} \boldsymbol{r} \vdash e : \tau$ *(using the rules in Figure 34) then there is a unique typing derivation using the typing rules from Figure 33 with a conclusion* $\Gamma \mathbin{@} \boldsymbol{r} \vdash e : \tau$ *obtained by induction over the original typing derivation as follows:*

Case (var)*,* (const)*: The resulting typing derivation uses the corresponding rule of the non-syntax-directed type system.*

Case (abs)*: Take the typing derivation for the sub-expression* e*. If the variable* x *does not appear in* Γ_1*, apply* (weak) *followed by* (abs)*. Otherwise assume* $\Gamma_1 = x_1 : \tau_1, \ldots, x_n : \tau_n$ *and* $x = x_i$*. Apply* (exch) *repeatedly on variables* x_i, x_{i+1} *then* x_i, x_{i+2} *and so on until it is applied on* x_i, x_n*. At this point,* x_i *is the last variable of the vector and we can apply* (abs)*. This produces the same consequent as the one in the original typing derivation.*

Case (app)*: Take the typing derivations for the sub-expressions* e_1 *and* e_2 *in free-variable contexts* Γ_1 *and* Γ_2*. For each variable* x *that appears in both* Γ_1 *and* Γ_2*, rename the variable to a fresh name* x' *in* e_1 *and to another fresh name* x'' *in* e_2 *and their typing derivations. Now we have disjoint contexts and we can apply* (app) *on the target derivations.*

Next, apply (exch) *until* x' *and* x'' *are last two variables in the vector and apply* (contr)*, renaming both* x' *and* x'' *to the original name* x*. Repeat this step for all variables that were renamed. The resulting variable context is* Γ *and the resulting coeffect annotation is the same as in the original typing derivation.*

6.4 SYNTACTIC PROPERTIES AND EXTENSIONS

When discussing the structural coeffect calculus in Section 6.2, we considered a language with variables, constants, application and abstraction. This lets us focus on the key properties of the coeffects, but it neglects a number of practical concerns. In this section, we extend the language with let binding and subcoeffecting. This is useful in practice, but it also shows other interesting aspects of the theory. Additional extensions that make the coeffect language practically useful are given by the implementation in Chapter 7.

6.4.1 *Let binding*

In the flat coeffect calculus, we included a special typing rule for let binding. As discussed in Section 4.5.2, this provides a more precise typing than the rule derived from abstraction and application, because it removes the ambiguity introduced by abstraction. For the structural coeffect system, the typing rule for let binding can be treated as a derived rule. The following shows the structural typing for `let`:

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

Thanks to the structural nature of the calculus, the coeffect $\mathbf{t}$ that is associated with the variable x is uniquely determined (as in function abstraction). It is then sequentially composed with the coeffects attached to the variables that actually appear in the sub-expression e_1.

Proposition 39 (Let binding). *In a structural coeffect calculus, the typing of* $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ *can be seen as a derived rule, i.e. its typing is equivalent to the typing of the expression* $(\lambda x.e_2)\ e_1$.

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.

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

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

6.4.2 *Subcoeffecting*

When discussing the flat coeffect calculus in Section 4.2.1, we noted that the $\leqslant$ operation for *flat coeffect algebra* can be defined in terms of $\oplus$ as follows:

$$\mathbf{r} \leqslant \mathbf{s} \iff \mathbf{r} \oplus \mathbf{s} = \mathbf{s}$$

This is not the case for the *structural coeffect scalar* structure. For example, in the calculus for tracking bounded reuse, the $\oplus$ operator is defined as $+$ (on integers) and $\leqslant$ is just $\leqslant$ and so the above equivalence does not hold. For this reason, we included $\leqslant$ as an explicit part of both of the structures.

The subcoeffecting rule is not syntax directed and we did not include it in the core calculus in order to keep the discussion about choosing unique derivation in Section 6.3 focused on the key problem – structural rules. The subcoeffecting rule for structural coeffect calculus looks as follows:

$$(sub)\ \frac{\Gamma @ \mathbf{r} \mathbin{+\!\!+} \langle \mathbf{s}' \rangle \mathbin{+\!\!+} \mathbf{q} \vdash e : \tau}{\Gamma @ \mathbf{r} \mathbin{+\!\!+} \langle \mathbf{s} \rangle \mathbin{+\!\!+} \mathbf{q} \vdash e : \tau}\ (\mathbf{s}' \leqslant \mathbf{s})$$

The sub-coeffecting is applied on individual variables rather than on the whole context, but it could be easily extended to a relation on vectors $\leqslant$. Subtyping on functions can be defined in exactly the same way as for the flat coeffect calculus (Section 4.5.1), because functions are annotated with a (single) coeffect scalar. It is worth noting that subcoeffecting is needed in Lemma 40 (discussed in the next section) when performing a substitution for a variable in an expression that does not contain the substituted variable.

6.5 SYNTACTIC EQUATIONAL THEORY

The properties of the structural coeffect algebra, together with two additional weak conditions, guarantee that certain equational properties on terms hold in all instances of the structural coeffect calculus that we consider in this thesis. In this section, we look at these common properties. In Section 6.5.1, we first briefly compare equational theory for flat coeffects (Section 4.4) and structural coeffects (Section 6.5.3).

6.5.1 *From flat coeffects to structural coeffects*

When discussing syntactic reductions for the flat calculus, we noted that call-by-name reduction does not, in general, preserve typing for all flat coeffect calculi. In the structural coeffect calculus, β-reduction and also η-expansion preserve typing for all instances of the calculus. Using the terminology of Pfenning and Davies [87], the structural coeffect calculus satisfies both the *local soundness* and the *local completeness* properties.

SUBSTITUTION FOR FLAT COEFFECTS (RECAP). The less obvious (*top-pointed*) variant of the substitution lemma for flat coeffects (Lemma 9) required all operations of the flat coeffect algebra to coincide. This enables substitution to preserve the type of expressions, because all additional demands arising as the result of the substitution can be associated with the declaration context. For example, consider the following example where Haskell-style implicit parameter ?offset is substituted for the variable y:

$$\begin{array}{rll} y : \mathsf{int} \mathbin{@} \emptyset \vdash \lambda x.y + ?\mathsf{total} & : \mathsf{int} \xrightarrow{\{?\mathsf{total}\}} \mathsf{int} & (\textit{before}) \\ () \mathbin{@} \{?\mathsf{offset}\} \vdash \lambda x.?\mathsf{offset} + ?\mathsf{total} & : \mathsf{int} \xrightarrow{\{?\mathsf{total}\}} \mathsf{int} & (\textit{after}) \end{array}$$

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 the typing rule for lambda abstraction – as all operators of the flat coeffect algebra used here are $\cup$, we can place the additional requirement on the outer context. Note that this is not the *only* possible typing, but it is a *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. By contrast, the substitution lemma (Lemma 40, page 137) for structural coeffects can be proven because 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 dataflow calculus (writing `prev` e to obtain previous value of the expression e) and we substitute $w + z$ for y:

$$\begin{array}{rll} y : \mathsf{int} \mathbin{@} \langle 2 \rangle \vdash \lambda x.\texttt{prev}\ (x + \texttt{prev}\ y) & : \mathsf{int} \xrightarrow{1} \mathsf{int} & (\textit{before}) \\ w : \mathsf{int}, z : \mathsf{int} \mathbin{@} 2 * \langle 1, 1 \rangle \vdash \lambda x.\texttt{prev}\ (x + \texttt{prev}\ (w + z))) & : \mathsf{int} \xrightarrow{1} \mathsf{int} & (\textit{after}) \\ w : \mathsf{int}, z : \mathsf{int} \mathbin{@} \langle 2, 2 \rangle \vdash \lambda x.\texttt{prev}\ (x + \texttt{prev}\ (w + z))) & : \mathsf{int} \xrightarrow{1} \mathsf{int} & (\textit{equivalently}) \end{array}$$

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 $*$ (which stands for the sequential composition $\circledast$ from the structural coeffect algebra) to the annotation of the new context – in the above example the coeffect $2 * \langle 1, 1 \rangle$ (*after*) is equivalent to the coeffect $\langle 2, 2 \rangle$ (*equivalently*).

6.5.2 *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. More importantly, because the system uses explicit contraction, we may also need to substitute for multiple variables in the variable context at the same time.

Consider the expression $\lambda x.x + x$. It is type-checked by type-checking $x_1 + x_2$, contracting x_1 and x_2 and then applying lambda abstraction. During the reduction of $(\lambda x.x + x)\ (y + z)$ we 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, such as the bunched typing system [73]. To express the substitution lemma later in this section, we follow the example of bunched type system and define the notion of a *context with holes*. A context with holes is a context such as $x_1 : \tau_1, \ldots, x_k : \tau_k \mathbin{@} \langle r_1, \ldots, r_k \rangle$, where some of the variable typings $x_i : \tau_i$ and corresponding coeffects r_i are replaced by *holes* written as $- \mathbin{@} -$.

Definition 13 (Context with holes). *We write $\Delta[- \mathbin{@} -]_n$ for a* context with n holes *(in addition to some number of variables). A context with holes is defined inductively over the number of holes:*

$$\begin{aligned}
\Delta[- \mathbin{@} -]_n &:= -, \Gamma \mathbin{@} \langle - \rangle \# \boldsymbol{s} && \text{where } \Gamma \mathbin{@} \boldsymbol{s} \in \Delta[- \mathbin{@} -]_{n-1} \\
\Delta[- \mathbin{@} -]_n &:= x : \tau, \Gamma \mathbin{@} \langle r \rangle \# \boldsymbol{s} && \text{where } \Gamma \mathbin{@} \boldsymbol{s} \in \Delta[- \mathbin{@} -]_n \\
\Delta[- \mathbin{@} -]_0 &:= () \mathbin{@} \langle \rangle
\end{aligned}$$

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 locations 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 14 (Hole filling). *Given a context with n holes $\Delta \mathbin{@} \boldsymbol{s} \in \Delta[- \mathbin{@} -]_n$, the hole filling operation written as $\Delta \mathbin{@} \boldsymbol{s}[\Gamma_1 \mathbin{@} \boldsymbol{r_1} \mid \ldots \mid \Gamma_n \mathbin{@} \boldsymbol{r_n}]$, replaces holes by the specified variables and corresponding coeffect annotations and is defined as:*

$$\begin{aligned}
-, \Delta \mathbin{@} \langle - \rangle \# \boldsymbol{s}\ [\Gamma_1 \mathbin{@} \boldsymbol{r_1} \mid \Gamma_2 \mathbin{@} \boldsymbol{r_2} \mid \ldots] &= \Gamma_1, \Gamma_2 \mathbin{@} \boldsymbol{r_1} \# \boldsymbol{r_2} \\
&\quad \text{where } \Gamma_2 \mathbin{@} \boldsymbol{r_2} = \Delta \mathbin{@} \boldsymbol{s}[\Gamma_2 \mathbin{@} \boldsymbol{r_2} \mid \ldots] \\
x_1 : \tau, \Delta \mathbin{@} \langle r_1 \rangle \# \boldsymbol{s}\ [\Gamma_1 \mathbin{@} \boldsymbol{r_1} \mid \Gamma_2 \mathbin{@} \boldsymbol{r_2} \mid \ldots] &= x_1 : \tau, \Gamma_2 \mathbin{@} \langle r_1 \rangle \# \boldsymbol{r_2} \\
&\quad \text{where } \Gamma_2 \mathbin{@} \boldsymbol{r_2} = \Delta \mathbin{@} \boldsymbol{s}[\Gamma_1 \mathbin{@} \boldsymbol{r_1} \mid \Gamma_2 \mathbin{@} \boldsymbol{r_2} \mid \ldots] \\
() \mathbin{@} \langle \rangle\ [\,] &= () \mathbin{@} \langle \rangle
\end{aligned}$$

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 9).

Lemma 40 (Multi-nary substitution). *In a structural coeffect calculus with a* structural coeffect scalar *such that $r \leqslant r' \Rightarrow \forall s.\ (r \circledast s) \leqslant (r' \circledast s)$ and also $\mathsf{ign} \leqslant (\mathsf{ign} \circledast r)$, given an expression with multiple holes that are filled by variables $x_1 : \tau_1, \ldots, x_n : \tau_n$ with coeffects $s_1, \ldots, s_n$:*

$$\Gamma \mathbin{@} \boldsymbol{r}\ [x_1 : \tau_1 \mathbin{@} \langle s_1 \rangle \mid \ldots \mid x_k : \tau_k \mathbin{@} \langle s_k \rangle] \vdash e_r : \tau_r$$

and a expressions e_i *with free-variable contexts* Γ_i *annotated with* $\mathbf{t_i}$*:*

$$\Gamma_1 @ \mathbf{t_1} \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma_k @ \mathbf{t_k} \vdash e_k : \tau_k$$

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* $s_i \circledast \mathbf{t_i}$*:*

$$\Gamma @ \mathbf{r}\,[\,\Gamma_1 @ s_1 \circledast \mathbf{t_1} \mid \ldots \mid \Gamma_k @ s_k \circledast \mathbf{t_k}\,] \vdash e_r[x_1 \leftarrow e_1] \ldots [x_k \leftarrow e_k] : \tau_r$$

Proof. By induction over $\vdash$, using the multi-nary aspect of the substitution in the proof of the contraction case (see Appendix B.2). □

The Lemma 40 has two additional requirements on the structural coeffect scalar that are similar to those of the substitution lemma for bottom-pointed flat coeffect systems (Lemma 9). Those two conditions are satisfied for all our examples (it would be reasonable to require them for *all* structural coeffect scalars, but we prefer to keep the original definition more general). The two requirements are needed in the proof for substitution for subcoeffecting rule (discussed in Section 6.4.2) and for weakening, respectively.

6.5.3 *Reduction and expansion*

In the Chapter 4, we discussed call-by-value separately from call-by-name, because the proof of call-by-value substitution has fewer prerequisites. In this section, we consider full β-reduction, which encompasses both call-by-value and call-by-name. We also show that η-expansion preserves the types. Both of the properties hold for a system with any structural coeffect algebra that satisfies the additional weak requirements given in Lemma 40.

REDUCTION THEOREM. In a full β-reduction, written as $\rightarrow_\beta$, we can replace the redex $(\lambda x.e_2)\ e_1$ by the expression $e_r[x \leftarrow e_s]$ anywhere inside a term. The subject reduction theorem guarantees that this does not change the type of the term.

Theorem 41 (Type preservation)**.** *In a structural coeffect system with subcoeffecting (Section 6.4.2) and a structural coeffect algebra formed by* $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leqslant)$ *and operations* $\langle - \rangle$ *and* $\circledast$ *that satisfies the requirements of Lemma 40, it holds that if* $\Gamma @ \mathbf{r} \vdash e : \tau$ *and* $e \rightarrow_\beta e'$ *using the full* β*-reduction then* $\Gamma @ \mathbf{r} \vdash e' : \tau$*.*

Proof. Consider the typing derivation for the redex $(\lambda x.e_r)\ e_s$:

$$\frac{\dfrac{}{\Gamma_s @ \mathbf{s} \vdash e_s : \tau_s} \qquad \dfrac{\Gamma_r, x{:}\tau_s @ \mathbf{r} \mathbin{\#} \langle \mathbf{t} \rangle \vdash e_r : \tau_r}{\Gamma_r @ \mathbf{r} \vdash \lambda x.e_r : \tau_s \xrightarrow{\mathbf{t}} \tau_r}}{\Gamma_r, \Gamma_s @ \mathbf{r} \mathbin{\#} (\mathbf{t} \circledast \mathbf{s}) \vdash (\lambda x.e_r)\ e_s : \tau_r}$$

For the substitution lemma, we first rewrite the typing judgement for e_r, i.e. $\Gamma_r, x{:}\tau_s @ \mathbf{r} \mathbin{\#} \langle \mathbf{t} \rangle \vdash e_r : \tau_r$ as a context with a single hole filled by the x variable: $\Gamma_r, - @ \mathbf{r} \mathbin{\#} - \ [\,x : \tau_s @ \langle \mathbf{t} \rangle\,] \vdash e_r : \tau_r$. Now we can perform the substitution using Lemma 40:

$$\frac{\dfrac{\Gamma_r, - @ \mathbf{r} \mathbin{\#} -\ [\,x{:}\tau_s @ \langle \mathbf{t} \rangle\,] \vdash e_r : \tau_r \qquad \Gamma_s @ \mathbf{s} \vdash e_s : \tau_s}{\Gamma_r, - @ \mathbf{r} \mathbin{\#} -\ [\,\Gamma_s @ \mathbf{t} \circledast \mathbf{s}\,] \vdash e_r[x \leftarrow e_s] : \tau_r}}{\Gamma_r, \Gamma_s @ \mathbf{r} \mathbin{\#} (\mathbf{t} \circledast \mathbf{s}) \vdash e_r[x \leftarrow e_s] : \tau_r}$$

The last step applies the hole filling operation, showing that substitution preserves the type of the term. □

Because of the vector structure of coeffect annotations $\mathbf{r}$, $\mathbf{s}$, and $\langle \mathbf{t} \rangle$, these are uniquely associated with Γ_r, Γ_s, and x respectively. Therefore, substituting

e_s (which has coeffects $\mathbf{s}$) for x introduces the context demands specified by $\mathbf{s}$ which are composed with the demands t associated with x, i. e. the variable being substituted.

EXPANSION THEOREM. Structural coeffect systems also exhibit η-equality, therefore satisfying both *local soundness* and *local completeness* as required by Pfenning and Davies [87]. Informally, this means that abstraction does not introduce too much, and application does not eliminate too much.

Theorem 42 (η-expansion)**.** *In a structural coeffect system with a structural coeffect algebra formed by* $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leqslant)$ *and operations* $\langle - \rangle$ *and* $\circledast$*, if* $\Gamma \mathbin{@} \boldsymbol{r} \vdash e : \tau$ *and* $e \to_\eta e'$ *using the full* η*-reduction then* $\Gamma \mathbin{@} \boldsymbol{r} \vdash e' : \tau$*.*

Proof. The following derivation shows that $\lambda x.f\ x$ has the same type and coeffects as the original expression f:

$$\dfrac{\dfrac{\dfrac{\Gamma \mathbin{@} \mathbf{r} \vdash f : \tau_1 \xrightarrow{s} \tau_2 \qquad x{:}\tau_1 \mathbin{@} \langle \mathsf{use} \rangle \vdash x : \tau_1}{\Gamma, x{:}\tau_1 \mathbin{@} \mathbf{r} \mathbin{+\!\!+} (s \circledast \langle \mathsf{use} \rangle) \vdash f\ x : \tau_2}}{\Gamma, x{:}\tau_1 \mathbin{@} \mathbf{r} \mathbin{+\!\!+} \langle s \rangle \vdash f\ x : \tau_2}}{\Gamma \mathbin{@} \mathbf{r} \vdash \lambda x.f\ x : \tau_1 \xrightarrow{s} \tau_2}$$

The second 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. □

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

6.6 CATEGORICAL MOTIVATION

To define the semantics of structural coeffect calculus, we follow the same approach as for flat coeffect calculus in Chapter 5. In this section, we define categorical semantics for the calculus in terms of *structural indexed comonad*, which is an extension of the *indexed comonad* structure. Similarly to *flat indexed comonad*, the structural variant adds operations that are needed to embed full λ-calculus, this time with per-variable contexts.

We use the semantics to guide the *categorically-inspired translation* discussed in Section 6.7, which translates context-aware programs from the structural coeffect calculus to a simple target functional language with uninterpreted comonadically-inspired primitives (that correspond to operations of the structural indexed comonads). We then give operational semantics for a concrete context-aware language by giving the domain-specific reduction rules for the comonadically-inspired primitives. As an example, we use this to prove syntactic type safety of structural dataflow language in Section 6.7.3.

6.6.1 *Semantics of vectors*

Recall that 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.

Previously, we defined the `map` function (in terms of `cobind` and `counit`), which transforms the value inside the context without affecting the coeffect annotation. Thus, we can use $\mathsf{map}_r\ \pi_i$ to transform a context containing product of variables $C^r(\tau_1 \times \ldots \times \tau_n)$ into a context containing a single 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 $C^{\langle r_1,\ldots,r_n\rangle}(\tau_1 \,\hat{\times}\, \ldots \,\hat{\times}\, \tau_n) \to \tau$. We assume that the operator is equipped with necessary associativity transformations allowing us to use it freely on more than two values.

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 *structural indexed comonad*, which operate over variable contexts contained in an indexed comonad C^r and are designed to preserve the correspondence between vectors and annotations.

In what follows, we model (finite) vectors of length n as $\tau_1 \,\hat{\times}\, \ldots \,\hat{\times}\, \tau_n$. As mentioned, we assume that the use of the operator can be freely reassociated. For example, when calling an operation that requires input of the form $(\tau_1 \,\hat{\times}\, \ldots \,\hat{\times}\, \tau_i) \,\hat{\times}\, (\tau_{i+1} \,\hat{\times}\, \ldots \,\hat{\times}\, \tau_n)$, we use an argument $(\tau_1 \,\hat{\times}\, \ldots \,\hat{\times}\, \tau_n)$ and assume that the appropriate transformation is inserted.

6.6.2 *Indexed comonads, revisited*

The semantics of structural coeffect calculus reuses the definition of *indexed comonad* with a minimal change. The additional structure that is required for context manipulation (merging and splitting) is different and is here provided by the *structural indexed comonad* structure that we introduce in this section.

Recall the definition from Section 5.2.4, 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 $C^r\tau_1 \to \tau_2$.

In the structural coeffect calculus, we work with morphisms of the form $C^r\tau_1 \to \tau_2$ 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 revisit the definition and use *coeffect vectors* in some of the operations.

Definition 15. *Given a monoid* $(\mathcal{C}, \circledast, \mathsf{use})$ *with a pointwise extension of the* $\circledast$ *operator to a vector (written as* $\mathtt{t} \circledast \mathtt{s}$*) and an operation lifting scalars to vectors* $\langle - \rangle$*, an* indexed comonad *over a category* $\mathcal{C}$ *is a triple* $(C^{r}, \mathsf{counit}_{\mathsf{use}}, \mathsf{cobind}_{s,r})$*:*

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

The object mapping C^{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 vector of length one. This means that it will always return a single value rather than a vector created using $- \hat{\times} -$. The $\mathsf{cobind}_{s,r}$ operation is, perhaps surprisingly, indexed by a coeffect vector and a coeffect scalar. This asymmetry is explained by the fact that the input function $(C^{r} \alpha \to \beta)$ 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 a vector containing just one value. In other words, α may contain $\hat{\times}$, but β may not, because the coeffect calculus has no way of constructing values containing $\hat{\times}$.

6.6.3 *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, the corresponding operations operate convert between a (wrapped) vector of values represented using $\hat{\times}$ and ordinary pairs of contexts containing parts of the vector. The vectors of coeffect annotations are split or merged using $+\!\!+$ of the structural coeffect algebra, in a way that mirrors the wrapped vectors (variable structure).

The following definition includes $\mathsf{dup}_{r,s}$ which models duplication of a variable in a context needed for the semantics of contraction:

Definition 16. *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* $\mathsf{merge}_{r,s}$*,* $\mathsf{split}_{r,s}$ *and* $\mathsf{dup}_{r,s}$ *where:*

- $\mathsf{merge}_{r,s}$ *is a family of mappings* $C^{r} \alpha \times C^{s} \beta \to C^{r +\!\!+ s} (\alpha \hat{\times} \beta)$
- $\mathsf{split}_{r,s}$ *is a family of mappings* $C^{r +\!\!+ s} (\alpha \hat{\times} \beta) \to C^{r} \alpha \times C^{s} \beta$
- $\mathsf{dup}_{r,s}$ *is a family of mappings* $C^{\langle r \oplus s \rangle} \alpha \to C^{\langle r, s \rangle} (\alpha \hat{\times} \alpha)$

Here, the following equalities must hold:

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

These 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-approximate 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 $r \oplus s$,

The semantics is defined over a typing derivation:

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

$$\frac{}{[\![() \mathbin{@} \langle\rangle \vdash n : \mathsf{num}]\!]} = \frac{}{\mathsf{const}\ n} \quad (\textit{num})$$

$$\frac{[\![\Gamma, x{:}\tau_1 \mathbin{@} \mathbf{r} \mathbin{+\!\!+} \langle s \rangle \vdash e : \tau_2]\!] = f}{[\![\Gamma \mathbin{@} \mathbf{r} \vdash \lambda x.e : \tau_1 \xrightarrow{s} \tau_2]\!] = f \circ \mathsf{curry}\ \mathsf{merge}_{\mathbf{r},\langle s \rangle}} \quad (\textit{abs})$$

$$\frac{\begin{array}{r}[\![\Gamma_1 \mathbin{@} \mathbf{r} \vdash e_1 : \tau_1 \xrightarrow{t} \tau_2]\!] = f \\ [\![\Gamma_2 \mathbin{@} \mathbf{s} \vdash e_2 : \tau_1]\!] = g\end{array}}{\begin{array}{r}[\![\Gamma_1, \Gamma_2 \mathbin{@} \mathbf{r} \mathbin{+\!\!+} (\mathbf{t} \circledast \mathbf{s}) \vdash e_1\ e_2 : \tau_2]\!] = \\ \mathsf{app} \circ f \times (\mathsf{cobind}_{\mathbf{t},\mathbf{s}}\ g) \\ \circ\ \mathsf{split}_{\mathbf{r},\mathbf{t} \circledast \mathbf{s}}\end{array}} \quad (\textit{app})$$

$$\frac{[\![\Gamma \mathbin{@} \mathbf{r} \vdash e : \tau]\!] = f}{[\![\Gamma, x{:}\tau_1 \mathbin{@} \mathbf{r} \mathbin{+\!\!+} \langle \mathsf{ign} \rangle \vdash e : \tau]\!] = f \circ \mathsf{snd} \circ \mathsf{split}_{\mathbf{r},\langle \mathsf{ign} \rangle}} \quad (\textit{weak})$$

$$\frac{\begin{array}{r}[\![\Gamma_1, x{:}\tau_1, y{:}\tau_2, \Gamma_2 \\ \mathbin{@} \mathbf{r} \mathbin{+\!\!+} \langle s, t \rangle \mathbin{+\!\!+} \mathbf{q} \vdash e : \tau]\!] = f\end{array}}{\begin{array}{r}[\![\Gamma_1, y{:}\tau_2, x{:}\tau_1, \Gamma_2 \\ \mathbin{@} \mathbf{r} \mathbin{+\!\!+} \langle t, s \rangle \mathbin{+\!\!+} \mathbf{q} \vdash e : \tau]\!] = \end{array} f \circ \mathsf{nest}_{\mathbf{r},\langle t,s \rangle,\langle s,t \rangle,\mathbf{q}} (\mathsf{merge}_{\langle s \rangle,\langle t \rangle} \circ \mathsf{swap} \circ \mathsf{split}_{\langle t \rangle,\langle s \rangle})} \quad (\textit{exch})$$

$$\frac{\begin{array}{r}[\![\Gamma_1, y{:}\tau_1, z{:}\tau_1, \Gamma_2 \\ \mathbin{@} \mathbf{r} \mathbin{+\!\!+} \langle s, t \rangle \mathbin{+\!\!+} \mathbf{q} \vdash e : \tau]\!] = f\end{array}}{\begin{array}{r}[\![\Gamma_1, x{:}\tau_1, \Gamma_2 \mathbin{@} \mathbf{r} \mathbin{+\!\!+} \langle s \oplus t \rangle \mathbin{+\!\!+} \mathbf{q} \\ \vdash e[z, y \leftarrow x] : \tau]\!] = \end{array} f \circ \mathsf{nest}_{\mathbf{r},\langle s \oplus t \rangle,\langle s,t \rangle,\mathbf{q}}\ \mathsf{dup}_{s,t}} \quad (\textit{contr})$$

Assuming the following auxiliary operations:

$$\mathsf{nest}_{\mathbf{r},\mathbf{s},\mathbf{s}',\mathbf{t}}\ f = \mathsf{merge}_{\mathbf{r},\mathbf{s}' \mathbin{+\!\!+} \mathbf{t}} \circ \mathsf{id} \times (\mathsf{merge}_{\mathbf{s}',\mathbf{t}} \circ f \times \mathsf{id} \circ \mathsf{split}_{\mathbf{s},\mathbf{t}}) \circ \mathsf{split}_{\mathbf{r},\mathbf{s} \mathbin{+\!\!+} \mathbf{t}}$$

$$\begin{array}{rcl}
\mathsf{id}\ x & = & x \\
\mathsf{const}\ v & = & \lambda x.v \\
\mathsf{curry}\ f\ x\ y & = & \lambda f.\lambda x.\lambda y.f\ (x, y) \\
\mathsf{fst}\ (x, y) & = & x \\
\mathsf{swap}\ (x, y) & = & (y, x) \\
f \times g & = & \lambda(x, y).(f\ x, g\ y) \\
\mathsf{app}\ (f, x) & = & f\ x
\end{array}$$

Figure 35: Categorical semantics of the structural coeffect calculus

duplicates the value of the variable α and splits the additional context between the two new variables. In a flat calculus, this operation was expressed using ordinary tuple construction, which is not possible here – the returned context needs to contain a two-element vector $\alpha \hat{\times} \alpha$.

6.6.4 *Semantics of structural calculus*

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. Additionally, 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 6.6.5 and start by looking at a unified categorical semantics defined in terms of *structural indexed comonads*.

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^{r}. Functions take only a single variable as an input and so the structure is annotated with a scalar, such as C^{r}, which we treat as being equivalent to a singleton vector annotation $C^{\langle r \rangle}$:

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

Note that the instances of flat indexed comonad ignored the fact that the variable context wrapped in the data structure is a product. This is not generally the case for the structural indexed comonads – the definitions shown in Section 6.6.5 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 to examine the structure of the variable context is another reason for using $-\hat{\times}-$ when interpreting expressions in contexts.

EXPRESSIONS. A semantics of structural coeffect calculi is shown in Figure 35. The semantics is written as composition of morphisms using a number of auxiliary definitions. Due to the equivalence between Cartesian Closed Categories and the λ-calculus, we will treat it as specifying translation to a target functional language in Section 6.7.

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 by a constant function.
- 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 $C^{\langle t \rangle} \tau_1 \to \tau_2$.

  A value $C^{\langle t \rangle} \tau_1$ required in order to call the function is obtained by applying $\mathsf{cobind}_{t,s}$ to the semantics of e_2. The result $C^{t \circledast s}(\ldots \hat{\times} \ldots \hat{\times} \ldots) \to C^{\langle t \rangle} \tau_1$ is then called with the latter part of the split input context.
- The semantics 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 (weakening, exchange, contraction) 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 an unused 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 $\mathsf{nest}_{r,s,s',t}$ 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 (*exch*) rule swaps two single-variable contexts and the (*contr*) rule uses the $\mathsf{dup}_{s,t}$ operation to duplicate a variable 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. The following correspondence theorem states that the annotations in the typing rules of the structural coeffect calculus correspond to the indices of the semantics. Thus, the calculus captures a context-dependent property if it can be modelled by a *structural indexed comonad*. As we show in the next section, this is the case for all three discussed examples (liveness, dataflow, bounded variable reuse).

Theorem 43 (Correspondence)**.** *In all of the typing rules of the structural coeffect system, the context annotations* r *and* s *of typing judgements* $\Gamma \mathbin{@} r \vdash e : \tau$ *and function types* $\tau_1 \xrightarrow{s} \tau_2$ *correspond to the indices of mappings* C^r *and* $\mathsf{C}^{\langle s \rangle}$ *in the corresponding semantic function defined by* $[\![\Gamma \mathbin{@} r \vdash e : \tau]\!]$.

Proof. By analysis of the semantic rules in Figure 35. □

6.6.5 *Examples of structural indexed comonads*

The categorical semantics for structural coeffect calculus is easily instantiated to give semantics for a concrete calculus. In this section, we revisit the three examples discussed throughout this chapter – structural liveness, dataflow and bounded variable reuse. Some aspects of the first two examples will be similar to flat versions discussed in Section 5.2.5 – 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 the parentheses, used for ordinary tuples.

DATAFLOW. It is interesting to note that the semantics of dataflow and bounded variable reuse (discussed next) 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 dataflow computations. For read-

ability, variables in bold face (such as $\mathbf{a_1}$) range over vectors while ordinary notation (such as a_1) is used for individual values.

Example 16 (Indexed list for dataflow)**.** *The indexed list model of dataflow 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 dataflow computations are defined as:

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

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

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

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 the 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 there represent options rather than lists).

The mappings that explain how dataflow 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 $\mathsf{dup}_{m,n}$ operation needs to produce a two-variable context containing m and n values, respectively, of the input variable. The input provides $max(m, n)$ values, so the definition is simply a matter of restriction. Finally, counit extracts the value of its single variable.

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 (see the definition of dup), the list now represents multiple copies of the same value as each value can only be accessed once. This semantics follows that of Girard [40].

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 Example* **??***. The operations that capture the behaviour of bounded reuse are:*

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

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

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

The counit operation is defined as previously – it extracts the only value of the only variable In the bounded variable reuse system, variable sharing is annotated with the $+$ operator (in contrast with *max* used in dataflow). The $\mathsf{dup}_{m,n}$ 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 dataflow and bounded reuse, the data type is defined as a vector of values obtained by applying the indexed list type constructor to types of individual variables. We can generalize this pattern. Given a parameterized (indexed) type constructor $D^l\alpha$, we define $C^{\langle l_1,\ldots,l_n\rangle}$ in terms of a vector of D^{l_i} types. For liveness, the definition lets us reuse some of the mappings 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 an appropriate dup operation.

Example 18 (Structural indexed option)**.** *Given a structural coeffect algebra formed by* $(\{L, D\}, \sqcap, \sqcup, L, D, \sqsubseteq)$ *and the indexed option data type* D^l*, such that* $D^D\alpha = 1$ *and* $D^L\alpha = \alpha$*, 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:*

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

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

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 $\cap$ 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. Depending on the required coeffect annotations, this may copy (duplicate) the value, or it may produce an empty context.

Finally, counit extracts a value which is always present as guaranteed by the type $C^{\langle \mathsf{L} \rangle}\,\alpha \to \alpha$. The lifting operation models subcoeffecting which can turn a context with a value into a dead context (second case); otherwise it behaves as identity.

PROPERTIES. The concrete categorical semantics presented in this section is a generalization of the concrete semantics given when introducing context-aware programming languages in Chapter 3.

Theorem 44 (Generalization)**.** *Consider a typing derivation obtained according to the rules for finding unique typing derivations as specified in Section 6.3 for a coeffect language with liveness, dataflow or bounded variable use.*

The semantics obtained by instantiating the rules in Figure 35 with the concrete operations defined in Example 18, Example 16 or Example 17 is the same as the one defined in Figure 19, Section 3.3.3 and Section 3.3.2, respectively.

Proof. Expansion of the definitions for the unique typing derivation. □

6.7 TRANSLATIONAL SEMANTICS

In the previous section, we used category theory to give a unified model capable of capturing the semantics of our three context-aware language based on the structural coeffect calculus. Although the categorical model is interesting on its own, we use it in the same way as in Chapter 5 – to define a translation from source context-aware languages to a simple target functional language. As for flat coeffect calculus, we show that the translation produces well-typed programs in the target language. For a sample context-aware programming language, we then show that well-typed programs (produced by the translation) do not get stuck.

This section mirrors the development presented in Chapter 5 for flat coeffect calculus. We extend the simple target language with additional constructs inspired by structural indexed comonads (Section 6.7.1), define the translation to the target language (Section 6.7.2) and prove safety of one sample language (Section 6.7.3) – we choose structural dataflow to allow easy comparison with the flat system.

6.7.1 *Comonadically-inspired language extensions*

In Section 5.3.1, we defined the syntax, typing rules and operational semantics for a simple functional programming language. We then extended it with uninterpreted constructs inspired by the *flat indexed comonad* structure and used it as the translation target for the flat coeffect calculus. In this section, we take the same core language and extend it with constructs inspired by the *structural indexed comonad*.

Given a coeffect language with a structural coeffect algebra formed by $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leqslant)$ and operations $\langle - \rangle$ and $\circledast, +\!\!+$, we extend the core functional language with operations shown in Figure 36. The syntax extensions

LANGUAGE SYNTAX. Given a structural coeffect algebra, extend the programming language syntax with the following constructs:

$$
\begin{array}{rcl}
e & = & \ldots \mid \mathsf{cobind}_{s,r}\ e_1\ e_2 \mid \mathsf{counit}_{\mathsf{use}}\ e \mid \mathsf{merge}_{r,s}\ e \mid \mathsf{split}_{r,s}\ e \mid \mathsf{dup}_{r,s}\ e \\
\tau & = & \ldots \mid C^{r}(\tau_1 \hat{\times} \ldots \hat{\times} \tau_k) \\
K & = & \ldots \mid \mathsf{cobind}_{s,r}\ \_\ e \mid \mathsf{cobind}_{s,r}\ v\ \_ \mid \mathsf{counit}_{\mathsf{use}}\ \_ \\
 & & \mid \mathsf{merge}_{r,s}\ \_ \mid \mathsf{split}_{r,s}\ \_ \mid \mathsf{dup}_{r,s}\ \_
\end{array}
$$

TYPING RULES. Given a structural coeffect algebra, add the typing rules:

$$
(\textit{counit})\ \frac{\Gamma \vdash e : C^{\langle \mathsf{use} \rangle}\tau}{\Gamma \vdash \mathsf{counit}_{\mathsf{use}}\ e : \tau}
$$

$$
(\textit{cobind})\ \frac{\Gamma \vdash e_1 : C^{r}(\tau_1 \hat{\times} \ldots \hat{\times} \tau_k) \to \tau \qquad \Gamma \vdash e_2 : C^{s \circledast r}\tau_1(\tau_1 \hat{\times} \ldots \hat{\times} \tau_k)}{\Gamma \vdash \mathsf{cobind}_{s,r}\ e_1\ e_2 : C^{\langle s \rangle}\tau}
$$

$$
(\textit{merge})\ \frac{\Gamma \vdash e : C^{r}(\tau_1 \hat{\times} \ldots \hat{\times} \tau_l) \times C^{s}(\tau_{l+1} \hat{\times} \ldots \hat{\times} \tau_k)}{\Gamma \vdash \mathsf{merge}_{r,s}\ e : C^{r+s}(\tau_1 \hat{\times} \ldots \hat{\times} \tau_k)}
$$

$$
(\textit{split})\ \frac{\Gamma \vdash e : C^{r+s}(\tau_1 \hat{\times} \ldots \hat{\times} \tau_k)}{\Gamma \vdash \mathsf{split}_{r,s}\ e : C^{r}(\tau_1 \hat{\times} \ldots \hat{\times} \tau_l) \times C^{s}(\tau_{l+1} \hat{\times} \ldots \hat{\times} \tau_k)}
$$

$$
(\textit{dup})\ \frac{\Gamma \vdash e : C^{\langle r \oplus s \rangle}\tau}{\Gamma \vdash \mathsf{dup}_{r,s}\ e : C^{\langle r,s \rangle}(\tau \hat{\times} \tau)}
$$

Figure 36: Comonadically-inspired extensions for structural coeffects

add comonadically-inspired operations that mirror those defined in Section 6.6.3. The typing for the operations corresponds to their categorical counterparts.

We also include an uninterpreted type $C^{r}\tau_1 \hat{\times} \ldots \hat{\times} \tau_k$, which models a contextual (comonadic) value indexed by a vector of annotations. As in the categorical model for structural coeffects, context consisting of multiple variables is not modelled as ordinary tuple – it can only be manipulated by the comonadically-inspired operations. In the target language, this is done by defining the C^{r} type over zero or more underlying types. Here, our syntactic treatment differs slightly from the categorical model where objects created by $\hat{\times}$ were first-class values. In Figure 36, we need to explicitly specify the types of individual components in typing rules for (*merge*), (*split*) and (*cobind*).

As with flat coeffects, the extensions described here are common for all concrete instances of structural context-aware languages. For each concrete language, we need to provide values of type $C^{r}\tau$ and reduction rules for comonadically-inspired operations.

6.7.2 *Comonadically-inspired translation*

When translating context-aware programs to the functional language, variable contexts become values of comonadically-inspired data types contain-

The translation is defined over a typing derivation:

$$\frac{}{[\![x{:}\tau \mathbin{@} \langle \mathsf{use} \rangle \vdash x : \tau]\!]} = \frac{}{\lambda ctx.\mathsf{counit}_{\mathsf{use}}\ ctx} \quad (var)$$

$$\frac{}{[\![() \mathbin{@} \langle\rangle \vdash n : \mathsf{num}]\!]} = \frac{}{\lambda ctx.n} \quad (num)$$

$$\frac{[\![\Gamma, x{:}\tau_1 \mathbin{@} \mathbf{r} \# \langle s \rangle \vdash e : \tau_2]\!] = \mathsf{f}}{[\![\Gamma \mathbin{@} \mathbf{r} \vdash \lambda x.e : \tau_1 \xrightarrow{s} \tau_2]\!] = \lambda ctx.\lambda v.\mathsf{f}\ (\mathsf{merge}_{\mathbf{r},\langle s \rangle}\ (ctx, v))} \quad (abs)$$

$$\frac{\begin{array}{r}[\![\Gamma_1 \mathbin{@} \mathbf{r} \vdash e_1 : \tau_1 \xrightarrow{t} \tau_2]\!] = \mathsf{f} \\ [\![\Gamma_2 \mathbin{@} \mathbf{s} \vdash e_2 : \tau_1]\!] = \mathsf{g}\end{array}}{\begin{array}{l}[\![\Gamma_1, \Gamma_2 \mathbin{@} \mathbf{r} \# (\mathbf{t} \circledast \mathbf{s}) \vdash e_1\ e_2 : \tau_2]\!] \\ \quad = \lambda ctx. \\ \qquad \mathsf{let}\ (ctx_1, ctx_2) = \mathsf{split}_{\mathbf{r},\mathbf{t} \circledast \mathbf{s}}\ ctx \\ \qquad \mathsf{f}\ ctx_1\ (\mathsf{cobind}_{\mathbf{t},\mathbf{s}}\ \mathsf{g}\ ctx_2)\end{array}} \quad (app)$$

$$\frac{[\![\Gamma \mathbin{@} \mathbf{r} \vdash e : \tau]\!] = \mathsf{f}}{\begin{array}{l}[\![\Gamma, x{:}\tau_1 \mathbin{@} \mathbf{r} \# \langle \mathsf{ign} \rangle \vdash e : \tau]\!] \\ \quad = \lambda ctx. \\ \qquad \mathsf{let}\ (ctx_1, \_) = \mathsf{split}_{\mathbf{r},\langle \mathsf{ign} \rangle}\ ctx \\ \qquad \mathsf{f}\ ctx_1\end{array}} \quad (weak)$$

$$\frac{\begin{array}{l}[\![\Gamma_1, x{:}\tau_1, y{:}\tau_2, \Gamma_2 \\ \quad \mathbin{@} \mathbf{r} \# \langle s, t \rangle \# \mathbf{q} \vdash e : \tau]\!]\end{array} = \mathsf{f}}{\begin{array}{l}[\![\Gamma_1, y{:}\tau_2, x{:}\tau_1, \Gamma_2 \\ \quad \mathbin{@} \mathbf{r} \# \langle t, s \rangle \# \mathbf{q} \vdash e : \tau]\!] \\ \quad = \lambda ctx.\ \mathsf{f}(\mathsf{nest}_{\mathbf{r},\langle t,s \rangle,\langle s,t \rangle,\mathbf{q}}\ (\lambda ctx'. \\ \qquad \mathsf{let}\ (ctx_1, ctx_2) = \mathsf{split}_{\langle t \rangle,\langle s \rangle} \\ \qquad \mathsf{merge}_{\langle s \rangle,\langle t \rangle}\ (ctx_2, ctx_1)\ ))\end{array}} \quad (exch)$$

$$\frac{\begin{array}{l}[\![\Gamma_1, y{:}\tau_1, z{:}\tau_1, \Gamma_2 \\ \quad \mathbin{@} \mathbf{r} \# \langle s, t \rangle \# \mathbf{q} \vdash e : \tau]\!]\end{array} = \mathsf{f}}{\begin{array}{l}[\![\Gamma_1, x{:}\tau_1, \Gamma_2 \mathbin{@} \mathbf{r} \# \langle s \oplus t \rangle \# \mathbf{q} \\ \quad \vdash e[z, y \leftarrow x] : \tau]\!] \\ \quad = \lambda ctx.\mathsf{f}\ (\mathsf{nest}_{\mathbf{r},\langle s \oplus t \rangle,\langle s,t \rangle,\mathbf{q}} \\ \qquad \mathsf{dup}_{s,t}\ ctx)\end{array}} \quad (contr)$$

Assuming the following auxiliary definition:

$$\begin{array}{l}\mathsf{nest}_{\mathbf{r},s,s',t} = \lambda \mathsf{f}.\lambda ctx. \\ \quad \mathsf{let}\ (ctx_1, ctx') = \mathsf{split}_{\mathbf{r},s \# t}\ ctx \\ \quad \mathsf{let}\ (ctx_2, ctx_3) = \mathsf{split}_{s,t}\ ctx' \\ \quad \mathsf{merge}_{\mathbf{r},s' \# t}\ (ctx_1, \mathsf{merge}_{s',t}\ (\mathsf{f}\ ctx_2, ctx_3))\end{array}$$

Figure 37: Translation from a structural coeffect calculus

ing a value for each variable in the context. Function inputs become comonadically-inspired values containing exactly one variable. More formally:

$$\begin{array}{rcl}[\![x_1{:}\tau_1, \ldots, x_n{:}\tau_n \mathbin{@} \langle r_1, \ldots, r_n \rangle]\!] & = & C^{\langle r_1,\ldots,r_n \rangle}([\![\tau_1]\!] \hat{\times} \ldots \hat{\times} [\![\tau_n]\!]) \\ [\![\tau_1 \xrightarrow{r} \tau_2]\!] & = & C^{\langle r \rangle}[\![\tau_1]\!] \to [\![\tau_2]\!] \\ [\![\mathsf{num}]\!] & = & \mathsf{num}\end{array}$$

The definition differs from the one for flat coeffects (Section 5.3.3) in that the comonadically-inspired data type takes multiple type parameters (separated using $\hat{\times}$), rather than wrapping a regular tuple in the target language.

The translation rules are defined in Figure 37. As in the case of the flat coeffect calculus, the definition directly follows the categorical semantics shown in Figure 37. We expand the definitions so that the result is a valid program in the target language rather than a composition of morphisms.

As with flat coeffects, the correspondence property of the semantics (Theorem 43) can now be adapted into well-typedness of the translation. Given a well-typed program in the structural coeffect calculus, the translation produces a well-typed program in the target language. This is true for all context-aware languages based on the structural coeffect calculus and it provides us with the first part of type safety theorem. The second part is type safety of the target language with concrete domain-specific extensions as discussed in Section 6.7.3.

Theorem 45 (Well-typedness of the translation)**.** *Given a typing derivation for a well-typed closed expression* $@\langle\rangle \vdash e : \tau$ *written in a structural context-aware programming language that is translated to the target language as (we write* $\ldots$ *for the omitted part of the translation tree):*

$$\frac{[\![\ (\ldots)\ ]\!]\ =\ (\ldots)}{[\![\, @\langle\rangle \vdash e : \tau]\!]\ =\ \mathsf{f}}$$

Then f *is well-typed, i. e. in the target language:* $\vdash \mathsf{f} : [\![() \, @ \langle\rangle]\!] \to [\![\tau]\!]$.

Proof. By rule induction over the derivation of the translation. Given a judgement $x_1 : \tau_1 \ldots x_n : \tau_n \, @ \, \mathbf{c} \vdash e : \tau$ where $\mathbf{c} = \langle c_1, \ldots, c_n \rangle$, the translation constructs a function of type $C^{\mathbf{c}}([\![\tau_1]\!] \hat{\times} \ldots \hat{\times} [\![\tau_n]\!]) \to [\![\tau]\!]$.

Case (*var*): $\mathbf{c} = \langle \mathsf{use} \rangle$ and so $\mathsf{counit}_{\mathsf{use}}$ *ctx* is well-typed.

Case (*num*): $\tau = \mathsf{num}$ and so the body n is well-typed.

Case (*abs*): The type of *ctx* is $C^{\mathbf{r}}(\ldots)$ and the type of v is $C^{\langle s \rangle}\tau_1$, calling $\mathsf{merge}_{\mathbf{r},\langle s \rangle}$ produces a context of type $C^{\mathbf{r} +\!\!+ \langle s \rangle}(\ldots \hat{\times} \tau_1)$ as expected by f.

Case (*app*): After applying $\mathsf{split}_{\mathbf{r},\mathbf{t} \circledast \mathbf{s}}$ the types of ctx_1, ctx_2 are $C^{\mathbf{r}}(\ldots)$ and $C^{\mathbf{t} \circledast \mathbf{s}}(\ldots)$, respectively. g requires $C^{\mathbf{s}}(\ldots)$ and so the result of $\mathsf{cobind}_{\mathbf{t},\mathbf{s}}$ is $C^{\langle \mathbf{t} \rangle}\tau_1$ as required by f.

Case (*weak*): After applying $\mathsf{split}_{\mathbf{r},\langle \mathsf{ign} \rangle}$ the type of ctx_1 is $C^{\mathbf{r}}(\ldots)$ as required.

Case (*exch*), (*contr*): The auxiliary definition $\mathsf{nest}_{\mathbf{r},\mathbf{s},\mathbf{s}',\mathbf{t}}$ keeps parts of the context corresponding to coeffect annotations $\mathbf{r}, \mathbf{q}$ unchanged and transforms the nested part. In (*exch*), the provided lambda function is of type $C^{\langle \mathbf{s},\mathbf{t} \rangle}(\tau_1 \hat{\times} \tau_2) \to C^{\langle \mathbf{t},\mathbf{s} \rangle}(\tau_2 \hat{\times} \tau_1)$. In (*contr*), the type of $\mathsf{dup}_{\mathbf{s},\mathbf{t}}$ is $C^{\langle \mathbf{s} \oplus \mathbf{t} \rangle}\tau \to C^{\langle \mathbf{s},\mathbf{t} \rangle}(\tau \hat{\times} \tau)$ (assuming nest is expanded, rather than treated as a value within the language). □

6.7.3 *Structural coeffect language for dataflow*

The target language with comonadically-inspired primitives provides a framework that can be used to model a variety of structural context-aware languages and prove their safety. As outlined in Section 5.5, the key principle that guarantees the safety of the target language is a correspondence between coeffect annotations and the values they represent. In a more expressive target language (such as Haskell or Agda), it would be sufficient to provide an implementation of the comonadically-inspired primitives for the concrete domain-specific language.

Our simple target language is not expressive enough to capture the correspondence in the type system and so we instead follow the same method-

LANGUAGE SYNTAX

$$\begin{array}{rcll} v & = & \ldots \mid \mathsf{Df}\langle \mathbf{v}_0, \ldots, \mathbf{v}_k \rangle & \mathbf{v} = \langle v_0, \ldots, v_k \rangle \\ e & = & \ldots \mid \mathsf{Df}\langle \mathbf{e}_0, \ldots, \mathbf{e}_k \rangle \mid \mathsf{prev}_{n_0,\ldots,n_k}\ e & \mathbf{e} = \langle e_0, \ldots, e_k \rangle \\ K & = & \ldots \mid \mathsf{Df}\langle \mathbf{v}_0, \ldots, \langle v_{j,0}, \ldots, v_{j,i-1}, \_, e_{j,i+1} \ldots, e_{j,n} \rangle, \ldots, \mathbf{e}_k \rangle & \\ & & \ldots \mid \mathsf{prev}_{n_0,\ldots,n_k}\ \_ & \end{array}$$

TYPING RULES

$$(vec)\ \frac{\forall i \in \{0 \ldots n\}.\ \Gamma \vdash e_i : \tau}{\Gamma \vdash_{\mathsf{vec}} \langle e_0, \ldots, e_n \rangle : \tau, n}$$

$$(df)\ \frac{\forall i \in \{0 \ldots k\}.\ \Gamma \vdash_{\mathsf{vec}} \mathbf{e}_i : \tau_i, n_i}{\Gamma \vdash \mathsf{Df}\langle \mathbf{e}_0, \ldots, \mathbf{e}_k \rangle : C^{\langle n_0,\ldots,n_k \rangle}(\tau_0 \hat{\times} \ldots \hat{\times} \tau_k)}$$

$$(prev)\ \frac{\Gamma \vdash e : C^{\langle n_0+1,\ldots,n_k+1 \rangle}(\tau_0 \hat{\times} \ldots \hat{\times} \tau_k)}{\Gamma \vdash \mathsf{prev}_{n_0,\ldots,n_k}\ e : C^{\langle n_0,\ldots,n_k \rangle}(\tau_0 \hat{\times} \ldots \hat{\times} \tau_k)}$$

TRANSLATION

$$\frac{[\![\Gamma @ \langle n_0 + 1, \ldots, n_k + 1 \rangle \vdash e : \tau]\!]}{[\![\Gamma @ \langle n_0, \ldots, n_k \rangle \vdash \mathsf{prev}\ e : \tau]\!]} = \frac{f}{\lambda ctx.\mathsf{prev}_{n_0,\ldots,n_k}\ ctx}$$

REDUCTION RULES

$$(counit)\quad \mathsf{counit}_0(\mathsf{Df}\langle\langle v_0 \rangle\rangle) \rightsquigarrow v_0$$

$$(cobind)\quad \begin{array}{l} \mathsf{cobind}_{m,\langle n_1,\ldots,n_k \rangle}\ f\ \mathsf{Df}\langle\langle a_{1,0}, \ldots a_{1,m+n_1} \rangle, \ldots, \langle a_{k,0}, \ldots a_{k,m+n_k} \rangle\rangle \rightsquigarrow \\ \quad \mathsf{Df}\langle\langle\ f(\mathsf{Df}\langle\langle a_{1,0}, \ldots a_{1,n_1} \rangle, \ldots, \langle a_{k,0}, \ldots a_{k,n_k} \rangle\rangle),\ \ldots\ , \\ \qquad f(\mathsf{Df}\langle\langle a_{1,m}, \ldots a_{1,m+n_1} \rangle, \ldots, \langle a_{k,m}, \ldots a_{k,m+n_k} \rangle\rangle)\ \ \rangle\rangle \end{array}$$

$$(merge)\quad \begin{array}{l} \mathsf{merge}_{\langle m_0,\ldots,m_k \rangle,\langle n_0,\ldots,n_l \rangle}((\mathsf{Df}\langle \mathbf{v}_0, \ldots, \mathbf{v}_k \rangle), (\mathsf{Df}\langle \mathbf{v}'_0, \ldots, \mathbf{v}'_l \rangle)) \rightsquigarrow \\ \quad \mathsf{Df}\langle \mathbf{v}_0, \ldots, \mathbf{v}_k, \mathbf{v}'_0, \ldots, \mathbf{v}'_l \rangle \end{array}$$

$$(split)\quad \begin{array}{l} \mathsf{split}_{\langle m_0,\ldots,m_k \rangle,\langle n_0,\ldots,n_l \rangle}(\mathsf{Df}\langle \mathbf{v}_0, \ldots, \mathbf{v}_k, \mathbf{v}'_0, \ldots, \mathbf{v}'_l \rangle) \rightsquigarrow \\ \quad (\mathsf{Df}\langle \mathbf{v}_0, \ldots, \mathbf{v}_k \rangle,\ \mathsf{Df}\langle \mathbf{v}'_0, \ldots, \mathbf{v}'_l \rangle) \end{array}$$

$$(prev)\quad \begin{array}{l} \mathsf{prev}_{\langle n_1,\ldots,n_k \rangle}(\mathsf{Df}\langle\langle v_{1,0}, \ldots, v_{1,n_1}, v_{1,n_1+1} \rangle, \ldots, \langle v_{k,0}, \ldots, v_{k,n_k}, v_{k,n_k+1} \rangle\rangle) \rightsquigarrow \\ \quad \mathsf{Df}\langle\langle v_{1,0}, \ldots, v_{1,n_1} \rangle, \ldots, \langle v_{k,0}, \ldots, v_{k,n_k} \rangle\rangle \end{array}$$

$$(dup)\quad \mathsf{dup}_{m,n}(\mathsf{Df}\langle\langle v_0, \ldots, v_{max(m,n)} \rangle\rangle) \rightsquigarrow \mathsf{Df}\langle\langle v_0, \ldots, v_m \rangle, \langle v_0, \ldots, v_n \rangle\rangle$$

Figure 38: Additional constructs for modelling structural dataflow

ology as when discussing safety for flat coeffect languages in section Section 5.4 and show safety for a sample concrete context-aware language. We consider the structural coeffect language for dataflow. The definitions can be compared with the flat version discussed in Section 5.4.1, which highlights the similarities and differences between the flat and structural notion of context.

DOMAIN-SPECIFIC EXTENSIONS. The Figure 38 extends the target functional language with constructs, typing rules, translation rules and reduction rules needed for modelling structural dataflow. In the structural model, the type $C^{\langle r_0,\ldots,r_n \rangle}(\tau_0 \hat{\times} \ldots \hat{\times} \tau_n)$ represents a structure that provides values and additional contexts for variables of types $\tau_0, \ldots, \tau_n$ with contextual capabilities as specified by corresponding coeffect annotations $r_0, \ldots, r_n$.

In case of dataflow, the comonadically-inspired structure keeps vectors of past values for each of the variables in the context. In the syntax, we write $\mathbf{v}$ and $\mathbf{e}$ for vectors of values and expressions, respectively. The expression $\mathsf{Df}\langle \mathbf{e}_0, \ldots, \mathbf{e}_k \rangle$ is then formed by a vector of variable assignments where each variable assignment is a vector of current and past values. When reducing expressions to values, we reduce values from left to right and from current-most value to the last past value. This is specified by the context K.

When type-checking expressions that create the Df values, we use an auxiliary judgement $\Gamma \vdash_{\mathsf{vec}} e : \tau, n$. The judgement checks that a vector of expressions $\mathbf{e}$ contains exactly n expressions of type τ. This is captured by the (*vec*) rule, which is then used to check individual elements of the context in the (*df*) rule.

PROPERTIES. Now consider a target language consisting of the core (ML-subset) defined by the syntax, reduction rules and typing rules given in Figure 28 (Chapter 5) with primitives inspired by structural indexed comonads defined in Figure 36 and also concrete notion of comonadically-inspired value and reduction rules for dataflow as defined in Figure 38.

As with the examples discussed in Chapter 5, the resulting language is type safe. Together with the well-typedness of the translation (Theorem 45), this guarantees type safety of the structural coeffect calculus for dataflow. In order to prove type safety, we first extend the *canonical forms lemma* (Lemma 17) and the *preservation under substitution lemma* (Lemma 18). Those need to consider the new (*df*) and (*prev*) typing rules and substitution under the newly introduced expression forms $\mathsf{Df}\langle \ldots \rangle$ and prev_n. We show that the translation rule for `prev` produces well-typed expressions. Finally, we extend the type preservation (Theorem 19) and progress (Theorem 20) theorems.

Theorem 46 (Well-typedness of the `prev` translation)**.** *Given a typing derivation for a well-typed closed expression* $@\langle\rangle \vdash e : \tau$*, the translated program* f *obtained using the rules in Figure 37 and Figure 38 is well-typed, i.e. in the target language:* $\vdash f : [\![() @ \langle\rangle]\!] \to [\![\tau]\!]$.

Proof. By rule induction over the derivation of the translation.

Case (*var, num, abs, app*): As before.

Case (*prev*): Type of *ctx* is $C^{\langle n_0+1,\ldots,n_k+1\rangle}(\tau_0 \hat{\times} \ldots \hat{\times} \tau_k)$ and so we can apply the (*prev*) rule to obtain $C^{\langle n_0,\ldots,n_k\rangle}(\tau_0 \hat{\times} \ldots \hat{\times} \tau_k)$ as required by f. □

Lemma 47 (Canonical forms)**.** *For all* e, τ*, if* $\vdash e : \tau$ *and* e *is a value then:*

1. *If* $\tau = \mathsf{num}$ *then* $e = n$ *for some* $n \in \mathbb{Z}$
2. *If* $\tau = \tau_1 \to \tau_2$ *then* $e = \lambda x.e'$ *for some* x, e'
3. *If* $\tau = \tau_1 \times \ldots \times \tau_n$ *then* $e = (v_1, \ldots, v_n)$ *for some* v_i
4. *If* $\tau = C^{\langle n_1,\ldots,n_k\rangle}(\tau_1 \hat{\times} \ldots \hat{\times} \tau_k)$ *then* $e = \mathsf{Df}\langle v_0, \ldots v_n \rangle$ *for some* v_i *such that* $v_i = \langle v_0, \ldots, v_{n_i} \rangle$.

Proof. (1,2,3) as before; for (4) the last typing rule must have been (*df*). □

Lemma 48 (Preservation under substitution)**.** *For all* $\Gamma, e, e', \tau, \tau'$*, if* $\Gamma, x : \tau \vdash e : \tau'$ *and* $\Gamma \vdash e' : \tau$ *then* $\Gamma \vdash e[x \leftarrow e'] : \tau$.

Proof. By induction over the derivation of $\Gamma, x : \tau \vdash e : \tau'$ as before, with new cases for $\mathsf{Df}\langle \ldots \rangle$ and prev_n. □

Theorem 49 (Type preservation). *If* $\Gamma \vdash e : \tau$ *and* $e \rightsquigarrow e'$ *then* $\Gamma \vdash e' : \tau$

Proof. Rule induction over $\rightsquigarrow$.

Case (*fn*, *prj*, *ctx*): As before, using Lemma 48 for (*fn*).

Case (*counit*): $e = \mathsf{counit}_0(\mathsf{Df}\langle\langle v_0\rangle\rangle)$. The last rule in the type derivation of e must have been (*counit*) with $\Gamma \vdash \mathsf{Df}\langle v_0\rangle\rangle : C^{\langle 0\rangle}\tau$ and therefore $\Gamma \vdash v_0 : \tau$.

Case (*cobind*): $e = \mathsf{cobind}_{m,\langle n_1,\ldots,n_k\rangle}\ f\ (\mathsf{Df}\langle \mathbf{v}_0, \ldots \mathbf{v}_k\rangle)$ such that $\forall i \in \{1 \ldots k\}$ $\mathbf{v}_i = \langle v_0, \ldots, v_{n_i}\rangle$. The last rule in the type derivation of e must have been (*cobind*) with a type $\tau = C^{\langle m\rangle}\tau'$ and assumptions:

- $\Gamma \vdash f : C^{\langle n_1,\ldots,n_k\rangle}(\tau_1 \hat{\times} \ldots \hat{\times} \tau_k) \to \tau_2$ and
- $\Gamma \vdash \mathsf{Df}\langle \mathbf{v}_0, \ldots \mathbf{v}_k\rangle : C^{\langle m+n_1,\ldots,m+n_k\rangle}(\tau_1 \hat{\times} \ldots \hat{\times} \tau_k)$

Using the (*df*) rule, the reduced expression has a type $C^{\langle m\rangle}\tau'$.

Case (*merge*, *split*, *next*): Similar. In all three cases, the last typing rule in the derivation of e guarantees that the context contains correct number of vectors and each vector contains a sufficient number of values of a correct type. □

Theorem 50 (Progress). *If* $\vdash e : \tau$ *then either* e *is a value or there exists* e' *such that* $e \rightsquigarrow e'$

Proof. By rule induction over $\vdash$.

Case (*num,abs,var,app,proj,tup*): As before, using the adapted canonical forms lemma (Lemma 47) for (*app*) and (*proj*).

Case (*counit*): $e = \mathsf{counit}_{\mathsf{use}}\ e_1$. If e_1 is not a value, it can be reduced using (*ctx*) with context $\mathsf{counit}_{\mathsf{use}}\ \_$, otherwise it is a value. From Lemma 47, $e_1 = \mathsf{Df}\langle\langle v\rangle\rangle$ and so we can apply (*counit*) reduction rule.

Case (*cobind*): $e = \mathsf{cobind}_{m,\langle n_1,\ldots,n_k\rangle}\ e_1\ e_2$. If e_1 is not a value, reduce using (*ctx*) with context $\mathsf{cobind}_{m,\langle n_1,\ldots,n_k\rangle}\ \_\ e$. If e_2 is not a value reduce using (*ctx*) with context $\mathsf{cobind}_{m,\langle n_1,\ldots,n_k\rangle}\ v\ \_$. If both are values then we have $e_2 = \mathsf{Df}\langle\langle a_{1,0}, \ldots a_{1,m+n_1}\rangle, \ldots, \langle a_{k,0}, \ldots a_{k,m+n_k}\rangle\rangle$ from Lemma 47 and so we can apply the (*cobind*) reduction.

Case (*merge*): $e = \mathsf{merge}_{\langle m_0,\ldots,m_k\rangle,\langle n_0,\ldots,n_l\rangle} e_1$. If e_1 is not a value, reduce using (*ctx*) with context $e = \mathsf{merge}_{\langle m_0,\ldots,m_k\rangle,\langle n_0,\ldots,n_l\rangle}\ \_$. If e_1 is a value, it must be a pair of vectors $(\mathsf{Df}\langle \mathbf{v}_0, \ldots, \mathbf{v}_k\rangle, \mathsf{Df}\langle \mathbf{v}'_0, \ldots, \mathbf{v}'_l\rangle)$ using Lemma 47 and it can reduce using (*merge*) reduction.

Case (*df*): $e = \mathsf{Df}\langle \mathbf{e}_0, \ldots, \mathbf{e}_n\rangle$. If $\mathbf{e}_i$ is not a value then reduce using (*ctx*) with the context $\mathsf{Df}\langle \ldots\rangle$. Otherwise, $\mathbf{e}_0, \ldots, \mathbf{e}_n$ are values and so $\mathsf{Df}\langle \mathbf{e}_0, \ldots, \mathbf{e}_n\rangle$ is also a value.

Case (*split*, *prev*): Similar. Either sub-expression is not a value, or the type guarantees that it is a stream with correct number of elements to enable the (*split*) or (*prev*) reduction, respectively. □

Theorem 51 (Safety of context-aware dataflow language). *If* $\Gamma \vdash e : \tau$ *and* $e \rightsquigarrow^* e'$ *then either* e' *is a value of type* τ *or there exists* e'' *such that* $e' \rightsquigarrow e''$ *and* $\Gamma \vdash e'' : \tau$.

Proof. Rule induction over $\rightsquigarrow^*$ using Theorem 49 and Theorem 50. □

6.8 SUMMARY

This chapter completes the key development of this thesis – the presentation of the coeffect framework, consisting of two calculi capturing properties of context-aware computations introduced in Chapter 3. In Chapters 4 and 5, we focused on whole-context properties of computations and we developed *flat coeffect calculus* to capture them. This chapter develops *structural coeffect calculus*, which captures *per-variable* contextual properties. The system provides a precise analysis of liveness and dataflow and allows other interesting uses such as tracking of variable accesses based on bounded linear logic.

Following the structure of the previous two chapters, the structural coeffect calculus is parameterized by a *structural coeffect algebra*. The two definitions are similar – both require operations $\circledast$ and $\oplus$ that model sequential and pointwise composition, respectively. For flat coeffects, we required $\wedge$ to model context merging. For structural coeffects, we instead use a vector (free monoid) with the $+\!\!+$ operation – which serves a similar purpose as $\wedge$. In order to keep track of separate annotations for each variable, we use a system with explicit structural rules (contraction, weakening and exchange) that manipulate the structure of variables and the structure of annotations at the same time.

The structural coeffect calculus has desirable equational properties that are satisfied only by certain flat coeffect calculi. In particular, we show that β-reduction and η-expansion have the typing preservation property for any instance of the structural coeffect calculus. These two strong properties are desirable for programming languages, but are often not satisfied (e. g. by languages with effects).

Finally, we discuss the semantics of the structural coeffect calculus in in terms of *structural indexed comonads*. As in Chapter 5, we first use the categorical semantics to unify the different notions of context (bounded reuse, dataflow and liveness) and then use it as a basis for translation that turns well-typed programs written in the structural coeffect calculus into well-typed programs of a simple functional language. We give concrete reduction rules for the target language, modelling *structural dataflow* and prove type safety of the language.

Part III

TOWARDS PRACTICAL COEFFECTS

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

In this part, we explore practical aspects of implementing context-aware programming languages based on coeffects and related future work. We discuss a prototype implementation (Chapter 7), which links together all parts of the theory discussed in the previous part. Building a production-ready programming language is outside the scope of the thesis, so we instead focus on conveying the concept of coeffects to broader audience and make the implementation available as a web-based interactive essay (Section 7.3) at: http://tomasp.net/coeffects.

In further work (Chapter 8), we outline unification of flat and structural coeffects (Section 8.1), which may be more suited for embedding in practical languages and we discuss alternative approaches for using coeffects in programming languages (8.2).

7 IMPLEMENTATION

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

The implementation directly follows the theory presented in Chapters 4, 5 and 6. It consists of a common framework that provides type checking and translation to a simple functional target language with comonadically-inspired primitives. Each concrete context-aware language then adds a domain-specific rule for choosing a unique typing derivation (as discussed in Section 4.3) together with a domain-specific definition of the comonadically-inspired primitives that define the runtime semantics (see Section 5.4).

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

CHAPTER STRUCTURE AND CONTRIBUTIONS

- We discuss how the implementation follows the theory (Section 7.1) presented earlier. This applies to the implementation of the *type checker* and the implementation of the *translation* to a simple target language that is then interpreted. We also discuss how the common framework makes it easy to implement additional context-aware languages (Section 7.1.3).
- We consider a number of case studies (Section 7.2) that illustrate interesting aspects of the theories discussed earlier. This includes the typing of lambda abstraction and the difference between flat and structural systems (Section 7.2.1) and the comonadically-inspired translation (Section 7.2.2).
- The implementation is available not just as downloadable code, but also in the format of interactive essay (Section 7.3), which aims to make coeffects accessible to a broader audience. We discuss the most interesting aspects of the web-based presentation and briefly discuss some of the interesting implementation detail (Section 7.3.2).

1 The *interactive essay* format is based on Bret Victor's work on *explorable explanations* [116] and is further explained in Section 7.3.1

7.1 FROM THEORY TO IMPLEMENTATION

The theory discussed so far provides the two key components of the implementation. In Chapter 4, we discussed the type checking of context-aware programs and Chapter 5 models the execution of context-aware programs (in terms of translation and operational semantics). For structural coeffects, the same components are discussed in Chapter 6. In this section, we discuss how these provide foundation for the implementation.

7.1.1 *Type checking and inference*

To simplify the writing of context-aware programs, the implementation provides a limited form of type inference (in contrast, previous chapters described just type *checking*). This is available for convenience. We do not claim any completeness result about the algorithm and we do not present full formalization. However, it is worth noting how the domain-specific procedures for choosing a unique type derivation (Section 4.3) are adapted.

The type inference works in the standard way [90, 28] by generating type constraints and solving them. Solving of type constraints is done in the standard way, but we additionally collect and solve *coeffect constraints*. In this section, we focus only on coeffect constraints (as generation and solving of type constraints is standard). The following (*abs*) rule demonstrates the notation used in this section:

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

The judgement $\Gamma \mathbin{@} r \vdash e : \tau \mid C$ denotes that an expression e in a context $\Gamma \mathbin{@} r$ has a type τ and produces coeffect constraints C. In the (*abs*) rule, we annotate the function body, function type and declaration site with new coeffect variables r, t and s, respectively and we generate a coeffect constraint $t = r \wedge s$ that captures the (*abs*) rule from flat coeffect calculus (Figure 22).

In order to obtain unique type derivation for each term (using the algorithms discussed in Section 4.3), we generate additional coeffect constraints for lambda abstraction of each flat coeffect language, as this is where flat coeffects permit multiple possible typings.

Example 19 (Flat implicit parameters.)**.** *As discussed in Section 4.3, when choosing unique typing derivation for implicit parameters, we keep track of the implicit parameters available in the lexical scope (written as Δ). In the lambda abstraction rule, the implicit parameters required by the body (tracked by r) are split so that all parameters available in lexical scope are captured and only the remaining parameters ($r \setminus \Delta$) are required from the caller of the function.*

From the presentation in Section 4.3, it might appear that resolving the ambiguity related to lambda abstraction for implicit parameters requires a type system different from the type system shown earlier in Section 4.2.2. This is not the case. We track implicit parameters in scope Δ, but the rest of the (abs) *rule from the implementation only generates an additional coeffect constraint. The adapted* (abs) *rule for implicit parameters looks as follows:*

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

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

Example 20 (Flat dataflow). *In a context-aware language for dataflow (and in language with liveness tracking), the inherent ambiguity of the* (abs) *rule is resolved by placing the context demands of the body on both the declaration site and the call site. In Section 4.3, this was defined by replacing the standard coeffect* (abs) *rule with a rule* (idabs) *that uses an annotation* r *for the body of the function, declaration site coeffect and call site coeffect.*

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

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

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

7.1.2 *Execution of context-aware programs*

Context-aware programs are executed by translating the source program into a simple functional target language with domain-specific primitives. For simplicity, programs in the simple target language are then interpreted, but they could equally be compiled using standard techniques for compiling functional code. The translation follows the rules defined in Section 5.3 (for flat coeffect languages) and Section 6.7 (for structural coeffect languages). The result of the translation is a program that consists of the following:

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

The fact that the prototype implementation is based on the theoretical framework provided by coeffect calculi means that it has the desirable properties proved in Section 5.4 and Section 6.7. In particular, evaluating a well-typed context-aware program in a context that provides sufficient contextual capabilities will not cause an error.

In the interactive essay (Section 7.3), we further use the coeffects to automatically generate a user interface that requires the user to provide the required contextual capabilities (past values for individual variables, or values for implicit parameters).

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

7.1.3 *Supporting additional context-aware languages*

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

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

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

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

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

The list mirrors a list of steps that need to be done when supporting a new effectful computation in a language that supports monadic "do" notation. The step (3) corresponds to implementing a new monad and (4) corresponds to adding monad-specific effectful operations. The step (2) applies when using indexing to track effects more precisely [79]. The only step that does not have a counterpart in effectful/monadic languages is (1).

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

7.2 CASE STUDIES

The prototype implementation illustrates a number of interesting aspects of coeffect systems. Those appear as examples in the interactive essay (discussed in Section 7.3), but we briefly review them in this section.

7.2.1 *Typing context-aware programs*

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

ABSTRACTION FOR IMPLICIT PARAMETERS. As discussed in Section 7.1.1, the implementation of the language with implicit parameters resolves the ambiguity in the lambda abstraction by generating a coeffect constraint that restricts the set of parameters required from the declaration site to those that are lexically available. Remaining parameters are required from the call site. This is illustrated by the following example:

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

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

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

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

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

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

7.2.2 *Comonadically-inspired translation*

In addition to running coeffect programs, the implementation can also display the result of the translation to the simple functional target language with comonadically-inspired primitives. The following two case studies illustrate important aspects of the translation for flat coeffect systems (Section 5.3) and structural coeffect systems (Section 6.7).

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

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

Translating the code to the target language produces the code below. The reader is encouraged to view the translation in the interactive essay (Section 7.3), which displays the types and coeffect annotations of the individual values and primitives. As in the theory, the comonadically-inspired primitives are families of operations indexed by the coeffects (we omit the annotations here):

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

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

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

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

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

```
fun x → fun y → prev x
```

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

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

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

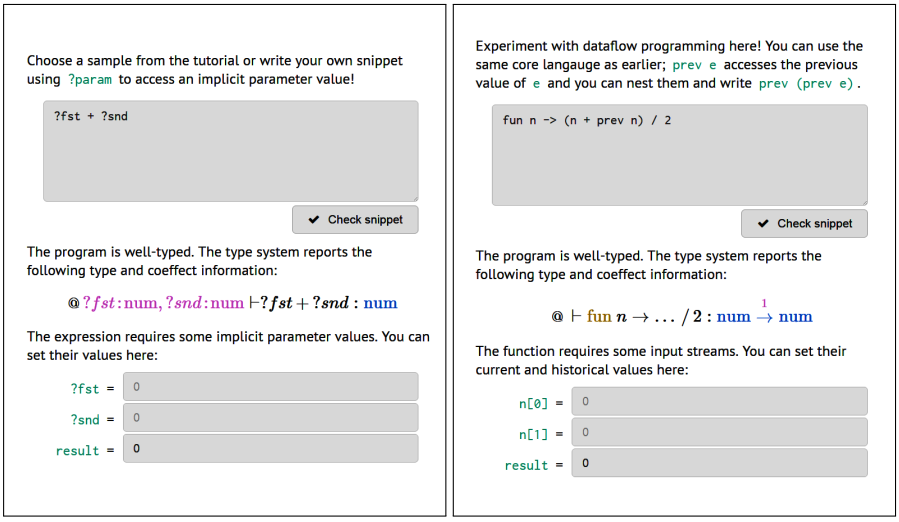

Figure 39: Interactive evaluation of implicit parameters (left) and dataflow (right)

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

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

7.3 INTERACTIVE ESSAY

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

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

To make the work *accessible*, we implement sample context-aware languages in a way that makes it possible to use them in any standard web browser with JavaScript support (Section 7.3.2) without requiring any server-side component. Following the idea that "the medium is the message" [64], we choose a medium that encourages *exploration* and make the implementation available not just as source code that can be compiled and run locally, but also in the format of an interactive essay (Section 7.3.1). The live version of the essay can be found at: http://tomasp.net/coeffects.

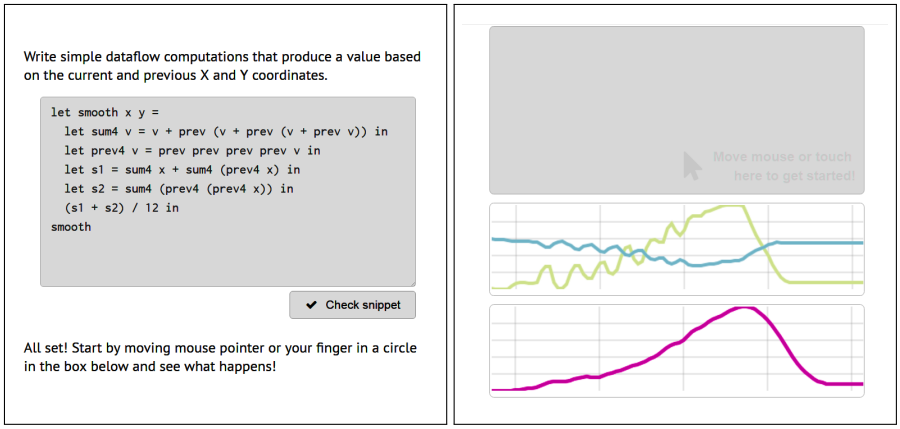

Figure 40: Function smoothing the X coordinate (left) with a sample run (right).

7.3.1 *Explorable language implementation*

The interactive essay format used of the implementation is based on Bret Victor's work on *explorable explanations* [116]. Bret Victor describes what the interactivity enables as follows:

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

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

The idea of active reading is older and has been encouraged in the context of art Josef Albers' classic 1963 work on color [4] (which was turned into an interactive essay 60 years later [89]). More recently, similar formats have been used to explain topics in areas such as signal processing [95] (explaining Fourier transformations) and sociology [46] (visualizing and explaining game theoretical model of segregation in the society [96]). To our best knowledge, the format has not previously been used in the area of programming language theory and so we briefly outline some of the interesting features that our essay provides in order to encourage active reading.

INTERACTIVE PROGRAM EXECUTION. After providing the practical motivation for coeffects (based on Chapter 1), the interactive essay shows the reader the two sample context-aware programming languages. Readers can write code in a panel that type checks the input, generates user interface for entering the required context and runs the sample code.

The panels for implicit parameters and for dataflow computations are shown in Figure 39. The sample code on the left adds two implicit parameters and the generated UI lets the user enter implicit parameter values as required by the context in the typing judgement. The sample on the right

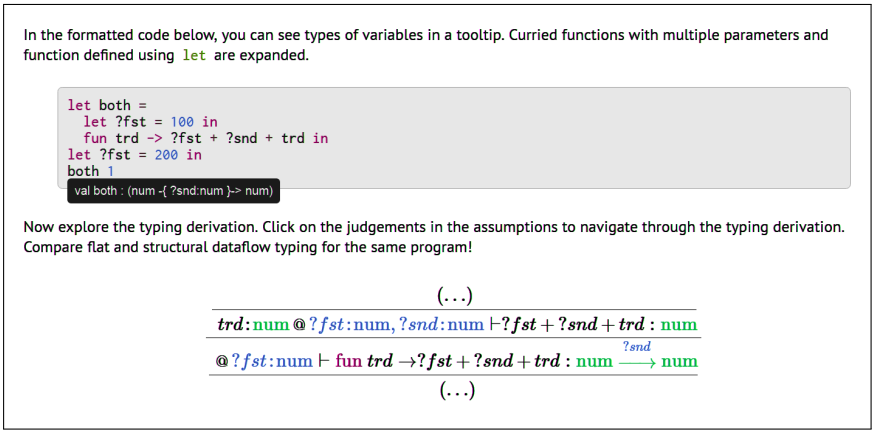

Figure 41: Source code with type information and explorable typing derivation.

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

The essay guides the readers through a number of programs (shown in Section 7.2.1) and encourages them to run and try modifying them. For implicit parameters, this includes the case where an implicit parameter is available at both declaration site and call site (showing that the declaration site value is captured). For dataflow, the examples include the comparison between the inferred type when using flat and structural type systems.

REACTIVE DATAFLOW. The interactive program execution lets the reader run sample programs, but not in a particularly realistic context. To show a more real-world scenario, the essay includes a widget shown in Figure 40. This lets the user write a function taking a stream of X and Y coordinates and calculate value based on the current and past values of the mouse pointer. The X and Y values, together with the result are plotted using a live chart.

In the example run shown above, the sample program calculates the average of the last 12 values of the X coordinate (green line in Figure 40). The example also illustrates one practical use of the coeffect type system – when running, the widget keeps the coordinates in a pre-allocated fixed-size array, because the coeffect type system guarantees that at most 12 past values will be accessed.

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

The presented interactive essay provides two features to help the reader understand and explore typing derivations. First, when reading code samples in the text, tooltips show the typing of individual identifiers, most interestingly functions (Figure 41, above).

Second, a later part of the essay provides a type checker that lets the user enter a source code in a context aware programming language and produces an explorable typing derivation for the program. The output (shown in Figure 41, below) displays a typing judgement with assumptions and conclu-

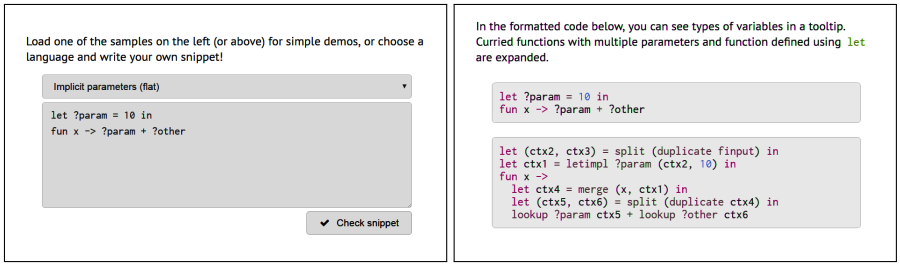

Figure 42: Source code with comonadically-inspired translation

sions and lets the reader navigate through the typing derivation by clicking on the assumptions or conclusions. This way, the reader can see how is the final typing derivation obtained, exploring interesting aspects, such as the abstraction rule (shown in Figure 41).

COMONADICALLY-INSPIRED TRANSLATION. The last interactive element of the essay lets the reader explore the translation of source context aware language to a target simple functional language (Section 7.1.2). Compared with the monadic "do" notation [50], the comonadic translation is more complex for two reasons.

First, it merges all variables into a single (comonadic) value representing the context. Second, there is a flat and structural variation of the system. For these two reasons, understanding the translation based just on the rules is harder than for monads. The essay lets readers see translation for carefully chosen case studies that illustrate the key aspects of the translation (some discussed in Section 7.2.2), but also for their own programs encouraging *active reading*. Sample input and output are shown in Figure 42.

7.3.2 *Implementation overview*

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

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

- The user interface of the interactive essay (Section 7.3.1) is implemented partly in F# and partly in JavaScript. The two most important components include a pretty-printer pretty.fs which formats source code (with type tooltips) and typing derivations and gui.fs that implements user interaction (e.g. navigation in explorable typing derivations).

As discussed in Section 7.1.3, the implementation can be easily extended to support additional context-aware programming languages. This is due to the fact that it is based on the unified theory of coeffects. In practice, adding support for liveness tracking would require adding a domain-specific constraint solver in solver.fs and extending the interpreter in evaluation.fs with a new kind of comonadically-inspired data type (indexed maybe comonad) with its associated operations.

7.4 RELATED WORK

To our best knowledge, combining Bret Victor's idea of explorable explanations with programming language theory (as discussed in Section 7.3) is a novel contribution of our work. On the technical side, we build on a number of standard methods.

PARSING AND TYPE CHECKING. The implementation of the parser, type checker and interpreter follows standard techniques for implementing functional programming languages. In order to be able to compile the implementation to JavaScript (see below), we built a small parser combinator library [49] rather than using one of the already available libraries [112].

TARGETTING JAVASCRIPT. In order to make the implementation accessible to a broad audience, it can be executed in a web browser. This is achieved by automatically translating the implementation from F# to JavaScript. We use an F# library called FunScript [14] (which is a more recent incarnation of the idea developed by the author [80]). We choose F#, but similar tools exist for other functional languages such as OCaml [119]. It is worth noting that FunScript is implemented as a *library* rather than as a compiler extension. This is done using the meta-programming capabilities of F# [102].

CLIENT-SIDE LIBRARY INTEGRATION. An interesting aspect of the interactive essay user interface is the integration with third-party JavaScript libraries. We use a number of libraries including JQuery (for web browser DOM manipulation) and MathJax [18] (for rendering of typing derivation). In order to call those from F# code, we use a number of mapping methods described in [86]. For example, the following F# declaration is used to invoke the Queue function in MathJax:

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

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

7.5 SUMMARY

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

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

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

8 FURTHER WORK

The prototype implementation discussed in the previous chapter directly follows the two coeffect calculi presented in this thesis. It links together all aspects of the coeffect theory (type system, semantics, safety proofs) and it provides practical support for the presented theoretical work. In this chapter, we consider alternative ways of embedding the theory of coeffects in practical programming languages and we also outline further directions in which the theory can be further developed.

The most important alternative directions revisit three design decisions made in this thesis. First, rather than discussing flat and structural coeffects, we can treat them in a single unified calculus tracking both kinds of context dependence (Section 8.1). Second, rather than using "language semantics" style (Section 2.3.1), we could use a "meta-language" style and extend the host language with explicit constructs for working with coeffects (Section 8.2). Third, rather than implementing a prototype coeffect language, we could embed coeffects into an existing language using techniques inspired by Haskell's "do" notation (Section 8.3).

CHAPTER STRUCTURE AND CONTRIBUTIONS

- We start by unifying the two kinds of coeffects discussed so far. We introduce the *unified coeffect calculus* (Section 8.1), which generalizes flat and structural systems and which can be instantiated to track both flat and structural properties.
- We discuss an alternative approach to defining coeffect systems based on the meta-language style (Section 8.2), which highlights the relationship between our work and related work arising from modal logics such as CMTT [70].
- We consider a way of implementing coeffects based on embedding comonadic computations via a lightweight syntactic extension and a rich type systems akin to Haskell's "do" notation (Section 8.3.1).
- Finally, we discuss extensions needed for real-world source languages with constructs such as conditionals and further applications of coeffects arising from substructural and bunched logics (Section 8.3.2)

The unified coeffect calculus presented in Section 8.1 is a novel result that completes the theory developed in this thesis and has been discussed in detail in a separate joint publication by the author [84]. We present it as further work here, because we do not supplement the unified calculus with the same operational treatment as the flat and structural coeffect calculi. Finally, the Sections 8.2 presents important related work and Section 8.3 presents important further work.

8.1 THE UNIFIED COEFFECT CALCULUS

The type systems of the flat coeffect calculus (Figure 22) and the structural coeffect calculus (Figure 33) differ in a number of ways. Understanding the differences is the key to reconciling the two systems:

- The structural coeffect calculus contains explicit rules for context manipulation (weakening, contraction, exchange). In the flat coeffect calculus, these rules are not explicit, but they can be derived thanks to the fact that variables in the context can be freely reordered. The (*var*) rule differs to permit implicit exchange and weakening in the flat system (by ignoring other variables in the context) and (*app*) implicitly permits contraction (by duplicating the context passed to sub-expressions).
- In the structural coeffect calculus, the variable context is treated as a vector and is annotated with a vector of (scalar) coeffects. In the flat coeffect calculus, the variable context is a finite partial function (from names to types) and is annotated with a single (scalar) coeffect.
- In the flat coeffect calculus (Figure 22), we distinguish between *splitting* context demands and *merging* of context demands ($\oplus$ and $\wedge$, respectively). In the structural coeffect calculus, the operations (which model appending and splitting vectors) are invertible and so the structural coeffect algebra uses the same tensor product $\#$. In essence $\oplus$ and $\wedge$ represent under- and over-approximation of $\#$. In the structural coeffect system, the $\oplus$ operation is still needed in the contraction rule.

In the unified calculus presented in this section, we address the three differences as follows. We include explicit rules for context manipulation in the calculus; in systems that arise from flat calculi, the structural rules do not change coeffects and can thus be applied freely. We generalize the structure of coeffect annotations using the notion of a "container" which can be specialized to obtain a single annotation or a vector of annotations. This could potentially be alternatively specialized to capture other structures such as trees in bunched typing [73]. Finally, we keep the $\oplus$ operation (needed in the contraction rule), but we distinguish between splitting and merging of context demands (using the notation $\perp\!\!\!\perp$ and $\top\!\!\!\top$, respectively). For structural coeffect calculi, the two operators coincide, but for flat coeffect calculi, they differ and provide the needed flexibility.

8.1.1 *Shapes and containers*

Our generalization of coeffect structure using a *coeffect container* is based on containers of Abbott et al. [2]. Containers have later been linked to comonads by Ahman et al. [3], but here we use them as part of the coeffect algebra, rather than to provide semantics of context-aware languages.

Intuitively a container describes data types such as lists, trees or streams. A container is formed by shapes (e. g. lengths of lists)[1]. For every shape, we can obtain a set of positions in the container (e. g. offsets in a list of a specified length). More formally:

Definition 17. *A* container *consists of a pair* (S, P)*, usually written as* $S \triangleleft P$*. Here,* S *is a set of shapes and* P *is a shape-indexed family of sets such that for each shape* $s \in S$*,* P_s *(also written as* $P(s)$*) is a set of positions for shape* s*.*

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

1 Shapes can be intuitively thought of as sizes, but this is not precise as some containers have multiple shapes of the same length, e. g. for perfectly ballanced, left- and right-leaning trees.

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

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

8.1.2 *Structure of coeffects*

In the structural coeffect calculus, coeffect annotations are formed by vectors of coeffect scalars. The annotations in the unified coeffect calculus are similar, but the notion of *vector* is replaced with a more general *container*. The primitive coeffect annotations in the unified calculus are formed by coeffect scalars, which have the same structure as in the structural coeffect calculus (Definition 11). In this section, we refer to it as *unified coeffect scalar* (we repeat the definition below for clarity). Then we define *unified coeffect containers* which determines how coeffect scalar values are attached to the free-variable context. Finally, we define the *unified coeffect algebra* which consists of shape-indexed coeffect scalar values.

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

COEFFECT SCALARS. The following definition of the coeffect scalar structure repeats Definition 11 from the Chapter 6, with the only change that we now refer to it as *unified coeffect scalar*.

Definition 18. *A unified coeffect scalar* $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leqslant)$ *is a set* $\mathcal{C}$ *together with elements* $\mathsf{use}, \mathsf{ign} \in \mathcal{C}$*, binary operations* $\circledast, \oplus$ *such that* $(\mathcal{C}, \circledast, \mathsf{use})$ *and* $(\mathcal{C}, \oplus, \mathsf{ign})$ *are monoids and a binary relation* $\leqslant$ *such that* $(\mathcal{C}, \leqslant)$ *is a pre-order. That is, for all* $r, s, t \in \mathcal{C}$*:*

$$r \circledast (s \circledast t) = (r \circledast s) \circledast t \qquad \mathsf{use} \circledast r = r = r \circledast \mathsf{use} \qquad \text{(monoid)}$$

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

$$\text{if } r \leqslant s \text{ and } s \leqslant t \text{ then } r \leqslant t \qquad t \leqslant t \qquad \text{(pre-order)}$$

In addition, the following distributivity axioms hold:

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

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

As previously, the monoid $(\mathcal{C}, \circledast, \mathsf{use})$ models sequential composition; the laws guarantee an underlying category structure; use and ign represent an accessed and unused variable, respectively. The $\oplus$ operation models combining of context demands arising from multiple parts of a program. Its meaning depends on the coeffect container. The operation can either combine requirements of individual variables (structural coeffects) or requirements attached to the whole context of multiple sub-expressions (flat coeffects).

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

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

In the coeffect judgment $\Gamma \mathbin{@} \mathbf{r} \vdash e : \tau$, the coeffect annotation $\mathbf{r}$ is drawn from the set of coeffect scalars $\mathcal{C}$ indexed by the shape corresponding to Γ. We write $l = \mathit{len}(\Gamma)$ for the shape corresponding to Γ and P_l is the *set* of positions for the given shape. A coeffect $\mathbf{r}$ can now be seen as a function $P_l \to \mathcal{C}$ that returns a scalar coefect $r \in \mathcal{C}$ for each of the position defined by the shape l. We write the set of positions for a shape l as $P(l)$ and a function that returns scalar coeffect as an exponent $\mathbf{r} \in \mathcal{C}^{P(l)}$.

Additionally, the coeffect container is equipped with an operation that appends shapes (when concatenating variable contexts) and two special shapes in S representing the empty context and the singleton context.

Definition 19. *A* coeffect container *structure* $(S \triangleleft P, \diamond, \hat{0}, \hat{1}, \mathit{len}(-))$ *comprises a container* $S \triangleleft P$ *with a binary operation* $\diamond$ *on* S *for appending shapes, a mapping from free-variable contexts to shapes* $\mathit{len}(\Gamma) \in S$*, and elements* $\hat{0}, \hat{1} \in S$ *such that* $(S, \diamond, \hat{0})$ *is a monoid.*

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

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

Example 21. *A structural coeffect container is defined as* $(S \triangleleft P, |\text{–}|, +, 0, 1)$ *where* $S = \mathbb{N}$ *and* $P(n) = \{1 \ldots n\}$*. The shape mapping* $|\Gamma|$ *returns the number of variables in* Γ*. Empty and singleton contexts are annotated with* 0 *and* 1*, respectively, and shapes of combined contexts are added so that* $|\Gamma_1, \Gamma_2| = |\Gamma_1| + |\Gamma_2|$*.*

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

Example 22. *A flat coeffect container is defined as* $(S \triangleleft P, |-|, \diamond, \star, \star)$*. The container is defined as a singleton data type* $S = \{\star\}$ *and* $P(\star) = \{0\}$ *with a constant function* $|\Gamma| = \star$ *and a trivial operation* $\star \diamond \star = \star$*.*

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

Example 23. *Similarly, we can also define a coeffect container with* no *positions, i. e.* $(S \triangleleft P, |-|, \diamond, \star, \star)$ *where* $S = \{\star\}$*,* $P(\star) = \emptyset$*,* $|\Gamma| = \star$ *and* $\star \diamond \star = \star$*. This reduces our system to the simply-typed* λ*-calculus with no context annotations, because* $P(\star) = \emptyset$ *and so coeffect annotations would be from the empty set* $\mathcal{C}^{\emptyset}$*.*

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

The definition here reconciles the third point discussed in Section 8.1 – the fact that flat coeffects use separate operations for splitting and merging ($\oplus$ and $\wedge$) while structural coeffects use the tensor $\#$. In the unified calculus, we use two operators that can, however, coincide.

Definition 20. *Given a unified coeffect scalar* $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leqslant)$ *and a coeffect container* $(S \triangleleft P, \mathit{len}(-), \diamond, \hat{0}, \hat{1})$ *a unified coeffect algebra extends the two structures with* $(\top\!\!\top, \perp\!\!\!\perp, \bot)$ *where* $\bot \in \mathcal{C}^{P(\hat{0})}$ *is a coeffect annotation for the empty context and* $\top\!\!\top, \perp\!\!\!\perp$ *are families of operations that combine coeffect annotations indexed by shapes. That is* $\forall n, m \in S$*:*

$$\begin{aligned} \perp\!\!\!\perp_{m,n} &: \mathcal{C}^{Pm} \times \mathcal{C}^{Pn} \to \mathcal{C}^{P(m \diamond n)} \\ \top\!\!\top_{m,n} &: \mathcal{C}^{Pm} \times \mathcal{C}^{Pn} \to \mathcal{C}^{P(m \diamond n)} \end{aligned}$$

A coeffect algebra induces the following three additional operations:

$$\begin{array}{lll} \langle - \rangle : \mathcal{C} \to \mathcal{C}^{P(\hat{1})} & \circledast_m : \mathcal{C} \times \mathcal{C}^{P(m)} \to \mathcal{C}^{P(m)} & \mathit{len}(-) : \mathcal{C}^{P(m)} \to m \\ \langle x \rangle = \lambda\_.x & r \circledast \mathbf{s} = \lambda s.r \circledast \mathbf{s}(s) & \mathit{len}(\mathbf{r}) = m \end{array}$$

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

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

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

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

The difference between $\top\!\!\top$ and $\perp\!\!\!\perp$ is clarified by the semantics [84], where $\mathbf{r} \top\!\!\top \mathbf{s}$ is an annotation of the *codomain* of a morphism that merges the capabilities provided by two contexts (in the syntactic reading, splits the context demands), while $\mathbf{r} \perp\!\!\!\perp \mathbf{s}$ is an annotation of the *domain* of a morphism that splits the capabilities of a single context into two parts (in the syntactic reading, merges their context demands). Syntactically, this means that we always use $\top\!\!\top$ in the rule *assumptions* and $\perp\!\!\!\perp$ in *conclusions*.

8.1.3 *Unified coeffect type system*

We define the unified coeffect system in Figure 43. It resembles the structural type system shown in Figure 33. Rather than explaining the rules one-by-one, we focus on how the unified system differs from the structural system.

The type system for the unified coeffect calculus is parameterized by a coeffect scalar $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leqslant)$ (Definition 18) together with a coeffect algebra $(\top\!\!\top, \perp\!\!\!\perp, \bot)$ (Definition 20) and the derived constructs $\langle - \rangle$, $\mathit{len}(-)$ and $\circledast$. As in the structural system, free-variable contexts Γ are treated as vectors

a.) Syntax-driven typing rules:

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

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

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

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

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

b.) Structural rules for context manipulation:

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

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

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

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

Figure 43: Type system for the unified coeffect calculus

of distinct variables with associativity built-in. The order of variables matters, but can be changed using the exchange rule. The context annotations $\mathbf{r}, \mathbf{s}, \mathbf{t}$ are shape-indexed coeffects (rather than simple vectors as before). As before, function arrows are annotated with coeffect scalars.

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

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

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

Note that we use the terms *merging* and *splitting* in the syntactic (top-down) sense. As discussed in Section 4.2.3, we can also read the rules in semantic (bottom-up) sense, in which case assumptions merge available contextual information and conclusions split available contextual information.

STRUCTURAL RULES. The merging/splitting operations in the structural rules are generalized in the same way as in the rules above. It is also worth noting that structural coeffect systems use vectors of elements while the unified system uses the container structure. For example, $\langle s, t\rangle$ and $\langle t, s\rangle$ in the (*exch*) rule denoted two-element vectors. In the unified system, this is replaced with merging/splitting of two lifted scalars: $\langle s\rangle \top \langle t\rangle$ and $\langle t\rangle \perp\!\!\!\perp \langle s\rangle$.

In structural systems, the two notations mean the same thing – we are simply concatenating or splitting two singleton vectors. However, this generalization allows us to capture flat coeffect systems as well. The lifting operation in flat systems simply returns the lifted scalar; operators $\top$ and $\perp\!\!\!\perp$ correspond to operations on coeffect scalars. As discussed in Section 8.1.5, thanks to the properties of coeffect scalars, the operations of contraction, weakening and exchange that do not affect the coeffect annotation are admissible for all flat systems embedded in the unified coeffect calculus.

8.1.4 *Structural coeffects*

The unified coeffect system uses a general notion of context shape, but it has been designed with structural and flat systems in mind. In this and the next section, we show how it captures the two coeffect systems from Chapter 4 and Chapter 6.

The unified calculus closely resembles the structural system and so using it to model structural systems is easy – given a (structural) coeffect scalar, we use the coeffect container that describes a *vector* of annotations (Example 21) and define a coeffect algebra formed by a vector (free monoid) of scalars.

Definition 21. *Given a coeffect scalar* $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leqslant)$ *a* structural coeffect system *is defined by:*

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

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

PROPERTIES. An important property of the unified system is that, when used in a structural way as discussed above, it gives calculi with the same properties as the structural system described in Chapter 6. This can be easily seen by comparing the Figure 43 with the Figure 33 and using the free monoid interpretation of the unified coeffect algebra.

Remark 52. *The system described in Definition 21 is equivalent to the structural coeffect system described in Figure 33. That is, a typing derivation using a structural coeffect embedded in the unified system is valid if and only if the corresponding derivation is valid in the structural system.*

Using the above definition, our unified coeffect system can capture per-variable coeffect properties discussed in Section 3.3. This includes the system for bounded reuse (which is only used in the structural form) and precise tracking of per-variable dataflow and liveness.

8.1.5 *Flat coeffects*

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

Flat coeffect systems are characterised by a singleton set of shapes (Example 22). In this setting, the context annotations $\mathcal{C}^{\mathsf{P}(\star)}$ are defined as function values $\{0\} \to \mathcal{C}$. The domain of the function is a singleton set and so the values are equivalent to coeffect scalars $\mathcal{C}$. In addition to the coeffect scalar structure, we need to define $\amalg$ and $\top\!\top$. Our examples of flat coeffects use $\oplus$ (merging of scalar coeffects) for $\amalg$ (merging of coeffect annotations). The $\top\!\top$ operation (corresponding to $\wedge$ in flat coeffect calculus) is provided explicitly.

Definition 22. *Given a coeffect scalar* $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leqslant)$ *and a binary operation* $\wedge : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ *such that* $(r \wedge s) \leqslant (r \oplus s)$*, the unified coeffect algebra modelling a flat coeffect systems consists of:*

- *A flat coeffect container* $(S \triangleleft \mathsf{P}, |-|, \diamond, \star, \star)$ *as defined in Example 22.*
- *A flat coeffect algebra* $(\wedge, \oplus, \mathsf{ign})$*, i.e.* $\amalg = \oplus$ *and* $\bot = \mathsf{ign}$ *with an additional binary operation* $\top\!\top = \wedge$*.*

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

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

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

Lemma 53. *A unified coeffect calculus capturing flat properties admits weakening that does not change the coeffect annotation.*

Proof. The rule is admissible as shown using the following derivation:

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

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

Lemma 54. *A unified coeffect calculus capturing flat properties admits exchange rule that does not change the coeffect annotation.*

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

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

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

Lemma 55. *A unified coeffect calculus capturing flat properties admits contraction rule that does not change the coeffect annotation.*

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

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

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

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

Lemma 56. *Weakening, as defined in a unified coeffect calculus capturing flat properties, is admissible in the flat coeffect calculus.*

Proof. Similar to the proof in Lemma 53. The property follows from the fact that $\bot = \langle \mathsf{ign} \rangle$ is the unit of $\oplus$. □

Lemma 57. *Contraction, as defined in a unified coeffect calculus capturing flat properties, is admissible in the flat coeffect calculus.*

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

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

From Definition 22, we know that $(r \wedge s \wedge t \wedge q) \leqslant (r \oplus s \oplus t \oplus q)$. Thus, the judgement can be derived using the (*sub*) rule of flat coeffect calculus. □

Lemma 58. *Exchange, as defined in a unified coeffect calculus capturing flat properties, is admissible in the flat coeffect calculus.*

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

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

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

A consequence of the equivalence discussed above is that the unified coeffect system can capture all properties that can be captured by the flat coeffect system – including implicit parameters, rebindable resources, Haskell type classes (discussed by Orchard [75]), dataflow and variable liveness.

8.2 COEFFECT META-LANGUAGE

In Section 2.3.1, we discussed two ways of using monads in programming language semantics as introduced by Moggi [67]. The first approach is to use monads in the *semantics* of an effectful language. The second approach is to extend the language with (additional) monadic constructs that can then be used for writing effectful monads explicitly.

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

In this section, we briefly discuss the other technique. That is, we embed indexed comonads into a λ-calculus as additional constructs of the meta-langauge. To do that, we introduce the type constructor $C^r\tau$ which represents a value τ wrapped in additional context (semantically, this corresponds to an indexed comonad) and we add language constructs that correspond to the operations of indexed comonads. The coeffect meta-language so derived highlights the relationship between coeffects and closely related work on contextual modal type theory (CMTT) [70]. Developing the system further is also an interesting future research direction.

8.2.1 *Coeffects and contextual modal type theory*

As discussed in Section 2.3.3, context-aware computations are related to modal logics – comonads have been used to model the □ modality and as a basis for meta-languages that include □ as a type constructor [11, 87, 11, 70]. Nanevski et al. [70] extend an S4 term language to a contextual modal type theory (CMTT). From the perspective of this thesis, CMTT can be viewed as a *meta-language* version of our coeffect calculus.

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

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

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

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

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

As discussed in Section 2.3.3, these signatures can be read in two different ways. One is the coeffect reading used in this thesis and another is the CMTT reading. The annotations have a different meaning in the two readings:

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

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

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

8.2.2 *Coeffect meta-language*

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

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

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

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

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

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

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

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

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

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

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

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

Figure 44: Type system for the (flat) coeffect meta-language

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

TYPES FOR COEFFECT META-LANGUAGE. The type system for the language is shown in Figure 44. The first part shows the usual typing rules for simply-typed λ-calculus. For simplicity, we omit typing rules for pairs, but those need to be present as the `merge` operation works on tuples.

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

The `split` and `merge` constructs follow a similar pattern. They both apply some transformation on one or two values in a context and then add the new value as a fresh variable to the variable context. We do not discuss subcoeffecting, but it could be easily added following the method used in Section 5.2.5.

3 The rule is similar to the letbox rule for ICML [70, p. 14], although it differs because of our generalization of comonads where bind composes coeffect annotations rather than requiring the same annotation everywhere.

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

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

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

Figure 45: Embedding the flat coeffect calculus in the coeffect meta-language

8.2.3 *Embedding flat coeffect calculus*

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

We first demonstrate that the flat coeffect calculus can be embedded in the meta-language described in the previous section. This may be desirable, e. g. when using the meta-language for reasoning about context-aware computations. We briefly consider this embedding as it illuminates the relationship between coeffects and CMTT. It is not possible to embed coeffect calculi in CMTT without generalisation such as the one presented here, because coeffect systems have a more general structure of annotations.

Given a typing judgement $\Gamma \mathbin{@} r \vdash e : \tau$ in the flat coeffect calculus, we define $[\![\Gamma \mathbin{@} r \vdash e : \tau]\!]_\nu$ as its embedding in the coeffect meta-language. The translation (Figure 45) is indexed by ν, which is a name of variable used to represent the entire variable context of the source language. The embedding resembles the semantics discussed in Section **??**. This is not surprising as the meta-language directly mirrors the operations of an indexed comonad.

8.3 RELATED AND FUTURE WORK

This section summarizes two interesting directions for future work and the related work in the area. First, we look at a way of embedding coeffects in existing languages such as Haskell that is inspired by the Haskell's "do" notation for monads (Section 8.3.1). Second, we discuss extensions to the theory of coeffects that are needed for real-world programming languages and we consider how the system could be extended to generalize languages based on substructural and bunched logics (Section 8.3.2).

8.3.1 *Embedded context-aware DSLs*

Many of the examples of contextual computation that we discussed earlier have been implemented as a single-purpose programming language feature (e. g. implicit parameters [60] or distributed computations [68, 25]). However, the main contribution of this thesis is that it captures *multiple* different notions of context-aware computations using just a *single* common abstraction. For this reason, future practical implementations of coeffects should not be single-purpose language features, but rather reusable abstractions that can be instantiated with a concrete *coeffect algebra* specified by the user. One approach towards this goal is to build a specialized context-aware lan-

guage such as the one implemented in Chapter 7. An alternative method is to embed context-aware DSLs into a richer target functional language.

In order to do this, the target programming language needs to provide two features; one that allows easiler embedding of context-aware computations themselves in programs akin to the "do" notation for monadsin Haskell and one that allows tracking of the contextual information in the type system.

EMBEDDING CONTEXTUAL COMPUTATIONS The embedding of *contextual* computations into programming languages can follow the successful model of *effectful* computations. In purely functional programming languages such as Haskell, effectful computations are embedded by *implementing* their model within the language and inserting the necessary (monadic) plumbing. This is made easier by the "do" notation or monad comprehensions [50, 39], which insert the monadic operations automatically.

The recently proposed "codo" notation [78] provides similar automation for context-aware computations based on comonads. The notation follows the semantics of our flat coeffect calculus (Chapter 4). Extending the "codo" notation to support calculi based on the structural coeffect calculus (Chapter 6) is interesting future work – this would require tracking the use of variables and the application of corresponding operations on the coeffect annotations, according to the structural rules.

In ML-like languages, effects (and many coeffects) are built-in into the language semantics, but they can still benefit from a special notation for explicitly marking effectful (coeffectful) blocks of code. In F#, this is done using *computation expressions* [85] that differ from the "do" notation in two ways. First, they support wider range of constructs, making it possible to wrap existing F# code within a block without other changes. Second, they support other abstractions including monads, applicative functors and monad transformers. It would be interesting to see if computation expressions can be extended to support programming with computations based on flat/structural indexed comonads.

More lightweight syntax for effectful computation can be obtained by automatically insertin the necessary monadic plumbing (without special syntax). This has been done for effectful computations in Lightewight ML [101] and a similar approach would be worth exploring for coeffects.

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

In a joint work on embedding effect systems in Haskell [79], we demonstrated that the recent additions to the Haskell type system provide enough power to implement structures such as sets at the type level. Using these to embed coeffect systems in Haskell is one fruitful future direction for implementing applied context-aware languages.

In dependently-typed languages such as Agda or Idris [12, 13], the embedding of coeffects can be implemented more directly (as terms implementing sets or lattices can be lifted to the type level). However, we believe that

coeffect tracking does not require full dependent types and can be made accessible in more mainstream languages. Dependent ML [128] provides an interesting example of a language with limited dependent typing (arithmetic) which is still close to its non-dependently-typed predecessor ML.

Another approach for embedding computations into the type system has been pioneered by F# *type providers* [103]. Technically, type providers are compiler extensions that generate types based on external data sources or other information in order to provide easy access to data or services. A similar approach could be used for embedding *algebras* such as coeffect algebras into the type system. An *algebra provider* would then be a library that specifies the objects of the algebra, its equational laws and rules for type inference. This could provide an easy-to-use way of embedding coeffect tracking in pragmatic languages such as F#. It is worth noting that the mechanism could also subsume F# units of measure [54]; these could then be seen as one such *algebra provider*.

8.3.2 *Extending the theory of coeffects*

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

Further work is to extend the coeffect systems presented here to a full programming language. A useful starting point is the work of Nielson and Nielson [71] who consider a similar development for effect systems, adding conditionals, recursion and polymorphism.

Coeffect annotations in context-aware programming languages can contain significant amount of information. Thus, the programming language also needs a form of type inference that can propagate such information. This has been done for effect systems [62]. For coeffects, the prototype implementation discussed in Chapter 7 provides basic support for type inference. This work has concentrated on the definition of coeffect systems and type checking; detailed treatment of type inference is future work.

LANGUAGE SEMANTICS. As discussed in Section 2.3, comonads can be used to define the semantics of a programming language in two ways. The first, "language semantics", approach is to use a single comonad to describe the semantics of a specific context-aware property. The second, "meta-language" approach is to extend the language with explicit constructs representing the operations of the underlying comonad.

The categorical semantics in this thesis used the "language semantics" style, but mainly as a guide for the development of the type systems. Further work includes more precise treatment of the categorical structure of indexed comonads. This has partly been done for flat coeffects by Orchard [75]. A joint work with Orchard [84] shows the first steps for structural systems. As discussed in Section **??**, further work is to develop a similar categorical model for the unified coeffect calculus, possibly using the categorical notion of containers [2].

We briefly outlined a calculus based on the "meta-language" approach in Section 8.2. Developing this technique further could unify coeffect systems with the work on Contextual Modal Type-Theory (CMTT) [70] and allow

other interesting applications of coeffects such as meta-programming [70] and distributed computing with explicit modalities [68].

SUBSTRUCTURAL AND BUNCHED LOGICS. In the *flat* and *structural* coeffect calculi, we attach annotations to the *entire context* and to *individual variables*, respectively. This chapter sketched unification of the two systems.

An intriguing question is whether coeffects can generalize *bunched typing* [73], which uses a tree-like structure of variable contexts (also discussed in Section 2.4). The current definition of the *unified coeffect calculus* is likely not expressive enough – bunched typing requires tree-like variable context, while we use a vector. Finding a simple coeffect system that is capable of capturing bunched typing is thus an interesting further work. Indeed, this could lead to numerous uses of coeffects as bunched typing is the basis for widely-used separation logic [74].

8.4 SUMMARY

In this chapter, we looked at three alternative directions for further development of coeffects. The key technical contribution of this chapter is the *unified coeffect system* (Section 8.1), which unifies the flat and structural system presented in the previous two chapters. This provides an alternative to the separate treatment of per-context and per-variable contextual dependencies at the cost of making the system more complex. To unify the two, we introduced *coeffect container*, which determines how are coeffect annotations attached to variable contexts. We then discussed two instances of the structure that capture flat and structural properties.

Next, we discussed how our work relates to meta-language based on comonads (Section 8.2). We present a *coeffect meta-language* that extends a simple functional programming language with additional language constructs corresponding to contextual operations. This is similar to the target languages used in the translational semantics in Chapter 5, but it follows a notation similar to the one used by contextual modal type theory (CMTT). This elucidates the relationship between our work and CMTT.

Finally, we considered an alternative approach to practical implementation of coeffect systems. Rather than building a specialized programming language (as we did in Chapter 7), we discuss the technical requirements for embedding coeffects as a domain-specific language (DSL) into a statically-typed functional language with a rich type system such as Haskell. This can be done by following the successful example of Haskell's "do" notation.

9 CONCLUSIONS

Some of the most fundamental academic work is not that which solves hard research problems, but that which changes how we understand the world. Some philosophers argue that *language* is the key for understanding how we think, while in science the dominant thinking is determined by *paradigms* [57] or *research programmes* [58]. Programming languages play a similar role for computer science and software development.

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

In this chapter, we give a brief overview of the technical contributions of this thesis (Section 9.1). The thesis looks at two kinds of context-dependence, identifies common patterns and captures those using two *coeffect calculi*. It gives a formal semantics using categorical insights and uses it as a basis for prototype implementation. Finally, Section 9.2 concludes the thesis.

9.1 CONTRIBUTIONS

As observed in Chapter 1, modern computer programs run in rich and diverse environments or *contexts*. The richness means that environments provides additional resources and capabilities that may be accessed by the program. The diversity means that programs often need to run in multiple different environments, such as mobile phones, servers, web browsers or even on the GPU. In this thesis, we present the foundations for building programming languages that simplify writing software for such rich and diverse environments.

NOTIONS OF CONTEXT. In λ-calculus, the term *context* is usually refers to the free-variable context. However, other programming language features are also related to context or program's execution environment. In Chapter 3, we revisit many of such features – including resource tracking in distributed computing, cross-platform development, dataflow programming and liveness analysis, but also Haskell's implicit parameters.

The main contribution of Chapter 3 is that it presents the disjoint language features in a unified way. We show type systems and semantics for many of the languages, illuminating the fact that they are closely related.

Considering applications is one way of approaching the theory of coeffects introduced in this thesis. Other pathways to coeffects are discussed in Chapter 2, which looks at theoretical developments leading to coeffects, including the work on effect systems, comonadic semantics and linear logics.

FLAT COEFFECT CALCULUS. The applications discussed in Chapter 3 fall into two categories. In the first category (Section 3.2), the additional contextual information are *whole-context* properties. They either capture properties of the execution environment or affect the whole free-variable context.

In Chapter 4, we develop a *flat coeffect calculus* which gives us an unified way of tracking *whole-context* properties. The calculus is parameterized by a *flat coeffect algebra* that captures the algebraic properties of contextual information. Concrete instances of flat coeffects include Haskell's implicit parameters, whole-context liveness and whole-context dataflow.

We define a type system for the calculus (Section 4.2), discuss how to resolve the ambiguity inherent in context-aware programming for sample languages we consider (Section 4.3) and study equational properties of the calculus (Section 4.4). In the flat coeffect calculus, β-reduction and η-expansion do not generally preserve the type of an expression, but we identify two conditions satisfied by two of our instances when this is the case.

STRUCTURAL COEFFECT CALCULUS. The second category of context-aware systems discussed in Section 3.3 captures *per-variable* contextual properties. The systems discussed here resemble substructural logics, but rather than *restricting* variable use, they *track* how the variables are used.

We characterise systems with *per-variable* contextual properties in Chapter 6, which describes our *structural coeffect calculus* (Section 6.2). Similarly to the flat variant, the calculus is parameterized by a *structural coeffect algebra*. Concrete instances of the calculus track bounded variable use (i. e. how many times is a variable accessed), dataflow properties (how many past values are needed) and liveness (i. e. can variable be accessed).

The structural coeffect calculus has desirable equational properties (Section 6.5). In particular, type preservation for β-reduction and η-expansion holds for all instances of the structural coeffect calculus. This follows from the fact that structural coeffects associate contextual requirements with individual variables and preserves the connection by including explicit structural rules (weakening, exchange and contraction).

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

This is done using a *comonadically-inspired* semantics. We generalize the category-theoretical dual of monad to *indexed comonad* and use it to define the semantics of context-aware programs for both the flat coeffect calculus (Chapter 5) and the structural coeffect calculus (Section 6.6). Due to the ambiguity inherent in contextual properties, we define the semantics over a typing derivation, but we also give a domain-specific algorithm for choosing a unique typing derivation for each of our sample languages (Section 4.3 and Section 6.3).

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

crete reduction rules need to be provided for each concrete context-aware domain-specific language. We give concrete definitions and prove type safety for three sample context-aware languages (Section 5.4 and Section 6.7). Together with well-typedness of our translation, these guarantee that "well-typed context-aware programs do not go wrong".

INTERACTIVE ESSAY. Associated with this essay is an interactive web-based essay, which implements three simple context-aware languages (Chapter 7). In addition to traditional prototype language implementations, the essay provides a number of novel features. As usual, it lets users write and evaluate sample programs, but it also lets them explore the details of the coeffect theory – this includes an interactive way of exploring typing derivations and program semantics. The essay also provides a number of case studies that highlight interesting aspects of the theory.

The implementation (Section **??**) serves two purposes. First, it links together all parts of the theory developed in this thesis (Section 7.2). It uses the flat and structural coeffect calculi to build a language framework that is then instantiated to implement three concrete context-aware languages. The framework provides the type system and translation (based on the comonadically-inspired semantics) that is used for program execution. This shows the practical benefits of using the theory of coeffects as a basis of real-world programming languages.

Second, web-based essay (http://tomasp.net/coeffects) discussed in Section 7.3 is novel from the pedagogical perspective. It makes our work *accessible* and *explorable*. Building a production-ready programming language is outside the scope of the thesis, so instead, our goal is to explain the theory and inspire authors of real-world programming languages to include support for context-aware programming, ideally using coeffects as a sound foundation. For this reason, we make the implementation accessible over web without requiring installation of specialized software and we make it explorable to encourage *active reading*.

9.2 SUMMARY

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

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

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

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

BIBLIOGRAPHY

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

[2] M. Abbott, T. Altenkirch, and N. Ghani. 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 International conference on foundations of software science and computational structures*, FOSSACS'12, pages 74–88, 2012.

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

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

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

[7] 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.

[8] 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.

[9] G. Biegel and V. Cahill. A framework for developing mobile, context-aware applications. In *Pervasive Computing and Communications, 2004*, pages 361–365. IEEE, 2004.

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

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

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

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

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

[15] F. Breuvart and M. Pagani. Modelling coeffects in the relational semantics of linear logic. In *Leibniz International Proceedings in Informatics*, volume 41. Schloss Dagstuhl, 2015.

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

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

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

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

[20] J. Cheney, A. Ahmed, and U. A. Acar. Provenance as dependency analysis. In *Proceedings of International conference on database programming languages*, DBPL'07, pages 138–152, Berlin, Heidelberg, 2007.

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

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

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

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

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

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

[27] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings of Symposium on principles of programming languages*, pages 262–275. ACM, 1999.

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

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

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

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

[32] A. Filinski. Towards a comprehensive theory of monadic effects. In *Proceeding of International conference on functional programming*, ICFP '11, pages 1–1, 2011.

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

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

[35] 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.

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

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

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

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

[40] 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.

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

[42] 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.

[43] 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.

[44] W. G. Halfond, A. Orso, and P. Manolios. Using positive tainting and syntax-aware evaluation to counter sql injection attacks. In *Proceedings of International symposium on foundations of software engineering*, pages 175–185. ACM, 2006.

[45] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of Symposium on principles and practice of parallel programming*, pages 48–60. ACM, 2005.

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

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

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

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

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

[51] P. Jouvelot and D. K. Gifford. Communication effects for message-based concurrency. Technical report, Massachusetts Institute of Technology, 1989.

[52] S.-y. Katsumata. Parametric effect monads and semantics of effect systems. In *Proceedings of Symposium on principles of programming languages*, POPL '14, pages 633–645, New York, NY, USA, 2014. ACM.

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

[54] A. J. Kennedy. Relational parametricity and units of measure. In *Proceedings of Symposium on principles of programming languages*, pages 442–455. ACM, 1997.

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

[56] G. A. Kildall. A unified approach to global program optimization. In *Proceedings of Symposium on principles of programming languages*, pages 194–206. ACM, 1973.

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

[58] I. Lakatos. *Methodology of scientific research programmes: philosophical papers*. Cambridge University Press.

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

[60] 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.

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

[62] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of Symposium on principles of programming languages*, POPL '88, pages 47–57, New York, NY, USA, 1988. ACM.

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

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

[65] E. Meijer, B. Beckman, and G. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *Proceedings of international conference on management of data*, SIGMOD '06, pages 706–706, 2006.

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

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

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

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

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

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

[72] D. L. Niki Vazou. Remarrying effects and monads. *Proceedings of MSFP*, 2014.

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

[74] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of International workshop on computer science logic*, CSL '01, pages 1–19, 2001.

[75] D. Orchard. Programming contextual computations. In *PhD thesis, University of Cambridge*.

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

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

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

[79] D. Orchard and T. Petricek. Embedding effect systems in Haskell. In *Proceedings Haskell Symposium*, Haskell '14, pages 13–24, 2014.

[80] T. Petricek. Client-side scripting using meta-programming. In *Bachelor thesis, Charles University in Prague*.

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

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

[83] 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.

[84] T. Petricek, D. Orchard, and A. Mycroft. Coeffects: A calculus of context-dependent computation. In *Proceedings of International conference on functional programming*, ICFP '14, pages 123–135, 2014.

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

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

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

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

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

[90] F. Pottier and D. Rémy. The essence of ML type inference. In *Advanced Topics in Types and Programming Languages*, 2005.

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

[92] A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in haskell. In *Proceedings of Haskell Symposium*, Haskell '08, pages 13–24, 2008.

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

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

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

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

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

[98] 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.

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

[100] 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.

[101] N. Swamy, N. Guts, D. Leijen, and M. Hicks. Lightweight monadic programming in ml. In *Proceedings of International conference on functional programming*, ICFP '11, pages 15–27, New York, NY, USA, 2011. ACM.

[102] 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.

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

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

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

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

[107] R. Tate. The sequential semantics of producer effect systems. In *Proceedings of Symposium on principles of programming languages*, POPL '13, pages 15–26, New York, NY, USA, 2013. ACM.

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

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

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

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

[112] S. Tolksdorf. FParsec – a parser combinator library for F#. Available at http://www.quanttec.com/fparsec, 2013.

[113] 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, 2005.

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

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

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

[117] 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*, volume 42, 2007.

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

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

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

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

[122] P. Wadler. Strictness analysis aids time analysis. In *Proceedings of Symposium on principles of programming languages*, pages 119–132, 1988.

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

[124] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of Symposium on principles of programming languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. ACM.

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

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

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

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

[129] B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving Haskell a promotion. In *Proceedings of Workshop on types in language design and implementation*, pages 53–66. ACM, 2012.

APPENDIX A

This appendix gives full typing derivations for the examples presented in Chapter 3, which demonstrate simple coeffect calculi for implicit parameters, liveness and dataflow.

A.1 COEFFECT TYPING FOR IMPLICIT PARAMETERS

The example on page 37 considers the typing of a function that adds the values of two implicit parameters: $\lambda x.?a + ?b$. Note that addition is a function (and so we write $(+)\ ?a\ ?b$) that requires no implicit parameters. Thus our typing derivation starts with:

$$\Gamma_0 = (+) : \mathsf{int} \xrightarrow{\emptyset} \mathsf{int} \xrightarrow{\emptyset} \mathsf{int}$$

The typing for the sub-expression $(+)\ ?a$ is shown in Figure 46 (a). Note that the resulting function does not require any implicit parameters, so there is no non-determinism so far. It is worth noting that this would change if we used η-expansion and wrote $(\lambda y.(+)\ ?a)$. We refer to the above as Δ. The rest of the expression is shown in Figure 46 (b).

Here, the last rule applies (*app*) and so we non-deterministically split the set of required resources. This means that we need r_1, r_2 such that $r_1 \cup r_2 = \{?a : \mathsf{int}, ?b : \mathsf{int}\}$. The Figure 46 (c) summarizes the 9 options.

A.2 COEFFECT TYPING FOR LIVENESS

The Section 3.2.3 discusses a coeffect system where the coeffect annotations capture whether a variable may be accessed (marked as live using L) or whether it is definitely not used (marked as dead using D). The following three examples are considered:

$$(\lambda x.42)\ y \qquad (1)$$
$$\mathsf{twoTimes}\ 42 \qquad (2)$$
$$(\lambda x.x)\ 42 \qquad (3)$$

The typing derivation for the expression (1) is shown in Figure 47 (a). The most interesting aspect about the previous example is the use of the (*app*) rule, which marks the resulting context as dead, even though a variable is accessed in the second part of the expression (this part never needs to be evaluated).

Assuming $\Gamma_0 = \mathsf{twoTimes} : \mathsf{int} \xrightarrow{\mathsf{L}} \mathsf{int}$, the typing derivation for (2) is shown in Figure 47 (b). Here, the variable context is marked as live. This is not because the argument of twoTimes is marked as live, but because the function itself is a variable that (always) needs to be obtained from the variable context. Finally, the derivation for (3) is shown in Figure 47 (c).

A.3 COEFFECT TYPING FOR DATAFLOW

The Section 3.2.4 presents a coeffect type system that tracks the maximal number of past values required by a dataflow computation. The discussion includes the following example:

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

In order to give typing for the above example, we first need to extend the language with conditionals. The typing rule for the `if` construct is:

$$(\mathit{if})\ \frac{\Gamma \circledast m \vdash e : \mathsf{bool} \qquad \Gamma \circledast n \vdash e_1 : \tau \qquad \Gamma \circledast n \vdash e_2 : \tau}{\Gamma \circledast \max(m, n) \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau_1}$$

Given a condition that requires m past and two (alternative) bodies that each require n past values, the composed expression requires at most the maximum of the two, i. e. the same context is passed to both the condition and the body using point-wise composition. As we will see, the fact that both branches require the same number of past values means that we need to use the subcoeffecting rule.

Assuming $\Gamma_0 = \mathsf{tick} : \mathsf{int}, \mathsf{counter} : \mathsf{int}$, the Figure 48 shows the typing derivation for the example. In the first part, we derive the types for the sub-expressions of the application and the conditional. Note that to obtain compatible function types, we use subcoeffecting *before* abstraction in the typing of $(\lambda x.x)$ – the type $\mathsf{int} \xrightarrow{1} \mathsf{int}$ is a valid overapproximation. Finally, the typing of the application yields the requirement of two past values, calculated as $\max(1, 1+1)$.

a.) Typing for the sub-expression $(+)\ ?a$

$$\dfrac{\dfrac{}{\Gamma_0, x : \mathsf{int} \mathbin{@} \emptyset \vdash (+) : \mathsf{int} \xrightarrow{\emptyset} \mathsf{int} \xrightarrow{\emptyset} \mathsf{int}}\ (var) \qquad \dfrac{}{\Gamma_0, x : \mathsf{int} \mathbin{@} \{?a : \mathsf{int}\} \vdash ?a : \mathsf{int}}\ (param)}{\Gamma_0, x : \mathsf{int} \mathbin{@} \{?a : \mathsf{int}\} \vdash (+)\ ?a : \mathsf{int} \xrightarrow{\emptyset} \mathsf{int}}\ (app)$$

b.) Typing for the expression $(+)\ ?a\ ?b$

$$\dfrac{\Delta \qquad \dfrac{\Gamma_0, x : \mathsf{int} \mathbin{@} \{?b : \mathsf{int}\} \vdash ?b : \mathsf{int}}{\Gamma_0, x : \mathsf{int} \mathbin{@} \{?a : \mathsf{int}, ?b : \mathsf{int}\} \vdash (+)\ ?a\ ?b : \mathsf{int}}\ (abs)}{\Gamma_0 \mathbin{@} r_1 \vdash \lambda x.(+)\ ?a\ ?b : \mathsf{int} \xrightarrow{r_2} \mathsf{int}}\ (app)$$

c.) Possible splittings of the implicit parameters

$r_1 = \{\}$	$r_2 = \{?a : \mathsf{int}, ?b : \mathsf{int}\}$
$r_1 = \{?a : \mathsf{int}\}$	$r_2 = \{?a : \mathsf{int}, ?b : \mathsf{int}\}$
$r_1 = \{?b : \mathsf{int}\}$	$r_2 = \{?a : \mathsf{int}, ?b : \mathsf{int}\}$
$r_1 = \{?a : \mathsf{int}, ?b : \mathsf{int}\}$	$r_2 = \{?a : \mathsf{int}, ?b : \mathsf{int}\}$
$r_1 = \{?a : \mathsf{int}, ?b : \mathsf{int}\}$	$r_2 = \{\}$
$r_1 = \{?a : \mathsf{int}, ?b : \mathsf{int}\}$	$r_2 = \{?a : \mathsf{int}\}$
$r_1 = \{?a : \mathsf{int}, ?b : \mathsf{int}\}$	$r_2 = \{?b : \mathsf{int}\}$
$r_1 = \{?a : \mathsf{int}\}$	$r_2 = \{?b : \mathsf{int}\}$
$r_1 = \{?b : \mathsf{int}\}$	$r_2 = \{?a : \mathsf{int}\}$

Figure 46: Coeffect typing for implicit parameters

a.) Typing for the expression $(\lambda x.42)\ y$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{y:\tau, x:\tau @ \mathsf{D} \vdash 42 : \mathsf{int}}\,(const)}{y:\tau @ \mathsf{D} \vdash \lambda x.42 : \tau \xrightarrow{\mathsf{D}} \mathsf{int}}\,(abs) \qquad \dfrac{}{y:\tau @ \mathsf{L} \vdash y : \tau}\,(var)}{y:\tau @ \mathsf{D} \sqcup (\mathsf{L} \sqcap \mathsf{D}) \vdash (\lambda x.42)\ y : \mathsf{int}}\,(app)}{y:\tau @ \mathsf{D} \vdash (\lambda x.42)\ y : \mathsf{int}}$$

b.) Typing for the expression $\mathsf{twoTimes}\ 42$

$$\dfrac{\dfrac{\dfrac{}{\Gamma_0 @ \mathsf{L} \vdash \mathsf{twoTimes} : \mathsf{int} \xrightarrow{\mathsf{L}} \mathsf{int}}\,(var) \qquad \dfrac{}{\Gamma_0 @ \mathsf{D} \vdash 42 : \mathsf{int}}\,(var)}{\Gamma_0 @ \mathsf{L} \sqcup (\mathsf{D} \sqcap \mathsf{L}) \vdash \mathsf{twoTimes}\ 42 : \mathsf{int}}\,(app)}{\Gamma_0 @ \mathsf{L} \vdash \mathsf{twoTimes}\ 42 : \mathsf{int}}$$

c.) Typing for the expression $(\lambda x.x)\ 42$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x:\mathsf{int} @ \mathsf{L} \vdash x : \mathsf{int}}\,(var)}{() @ \mathsf{L} \vdash \lambda x.x : \mathsf{int} \xrightarrow{\mathsf{L}} \mathsf{int}}\,(abs) \qquad \dfrac{}{() @ \mathsf{D} \vdash 42 : \mathsf{int}}\,(const)}{() @ \mathsf{L} \sqcup (\mathsf{L} \sqcap \mathsf{D}) \vdash (\lambda x.x)\ 42 : \mathsf{int}}\,(app)}{() @ \mathsf{L} \vdash (\lambda x.x)\ 42 : \mathsf{int}}$$

Figure 47: Coeffect typing for liveness

a.) Typing for sub-expressions of the conditional and for the argument

$$(prev)\ \frac{\dfrac{\Gamma_0 \circledast 0 \vdash \mathsf{tick} : \mathsf{int}}{\Gamma_0 \circledast 1 \vdash \mathtt{prev}\ \mathsf{tick} : \mathsf{int}}}{\Gamma_0 \circledast 1 \vdash (\mathtt{prev}\ \mathsf{tick}) = 0 : \mathsf{bool}}$$

$$(prev)\ \frac{(var)\ \dfrac{}{\Gamma_0, x{:}\mathsf{int} \circledast 0 \vdash x : \mathsf{int}}}{(abs)\ \dfrac{\Gamma_0, x{:}\mathsf{int} \circledast 1 \vdash \mathtt{prev}\ x : \mathsf{int}}{\Gamma_0 \circledast 1 \vdash \lambda x.\mathtt{prev}\ x : \mathsf{int} \xrightarrow{1} \mathsf{int}}}$$

$$(abs)\ \frac{(sub)\ \dfrac{(var)\ \dfrac{}{\Gamma_0, x{:}\mathsf{int} \circledast 0 \vdash x : \mathsf{int}}}{\Gamma_0, x{:}\mathsf{int} \circledast 1 \vdash x : \mathsf{int}}}{\Gamma_0 \circledast 0 \vdash \lambda x.x : \mathsf{int} \xrightarrow{1} \mathsf{int}}$$

$$(prev)\ \frac{(var)\ \dfrac{}{\Gamma_0 \circledast 0 \vdash \mathsf{counter} : \mathsf{int}}}{\Gamma_0 \circledast 1 \vdash \mathtt{prev}\ \mathsf{counter} : \mathsf{int}}$$

b.) Typing for the composed expression

$$(if)\ \frac{(app)\ \dfrac{(\ldots)}{\Gamma_0 \circledast 1 \vdash (\mathtt{if}\ldots\mathtt{then}\ldots\mathtt{else}\ldots) : \mathsf{int} \xrightarrow{1} \mathsf{int}}}{\dfrac{\Gamma_0 \circledast \max(1, 1+1) \vdash (\mathtt{if}\ldots\mathtt{then}\ldots\mathtt{else}\ldots)\ (\ldots) : \mathsf{int}}{\Gamma_0 \circledast 2 \vdash (\mathtt{if}\ldots\mathtt{then}\ldots\mathtt{else}\ldots)\ (\ldots) : \mathsf{int}}} \qquad \Gamma_0 \circledast 1 \vdash (\ldots) : \mathsf{int}$$

Figure 48: Coeffect typing for dataflow

APPENDIX B

This appendix provides additional details for some of the proofs for equational theory of flat coeffect calculus from Chapter 4 and structural coeffect calculus from Chapter 6.

B.1 SUBSTITUTION FOR FLAT COEFFECTS

In Section 4.4.3, we stated that, for a bottom-pointed flat coeffect algebra (i.e. $\forall r \in \mathcal{C} \,.\, r \geqslant \mathsf{use}$), the call-by-name substitution preserves type if all operators of the flat coeffect algebra coincide (Lemma 9). This section provides the corresponding proof.

Lemma (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 and* $r \leqslant r' \Rightarrow \forall s. r \circledast s \leqslant r' \circledast s$ *then:*

$$\Gamma \mathbin{@} S \vdash e_s : \tau_s \quad \wedge \quad \Gamma_1, x : \tau_s, \Gamma_2 \mathbin{@} r \vdash e_r : \tau_r$$
$$\Rightarrow \quad \Gamma_1, \Gamma, \Gamma_2 \mathbin{@} r \circledast S \vdash e_r[x \leftarrow e_s] : \tau_r$$

Proof. Assume that $\Gamma \mathbin{@} S \vdash e_s : \tau_s$ and we are substituting a term e_s for a variable x. Note that we use upper-case S to distinguish the coeffect of the expression that is being substituted into an expression. Using structural induction over $\vdash$:

(VAR) Given the following derivation using (*var*):

$$\frac{}{\Gamma_1, y : \tau, \Gamma_2 \mathbin{@} \mathsf{use} \vdash y : \tau}$$

There are two cases depending on whether y is the variable x or not:

- If $y = x$ then also $\tau = \tau_s$ and thus $y[x \leftarrow e_s] = e_s$. Using the assumption, implicit weakening and the fact that use is a unit of $\circledast$:

$$\frac{\dfrac{\Gamma \mathbin{@} S \vdash y[x \leftarrow e_s] : \tau_s}{\Gamma_1, \Gamma, \Gamma_2 \mathbin{@} S \vdash y[x \leftarrow e_s] : \tau}}{\Gamma_1, \Gamma, \Gamma_2 \mathbin{@} \mathsf{use} \circledast S \vdash y[x \leftarrow e_s] : \tau}$$

- If $y \neq x$ then $y[x \leftarrow e_s] = y$. Using the fact that use is the bottom element and subcoeffecting:

$$\frac{\Gamma_1, y : \tau, \Gamma_2 \mathbin{@} \mathsf{use} \vdash y : \tau}{\Gamma_1, y : \tau, \Gamma_2 \mathbin{@} \mathsf{use} \circledast S \vdash y : \tau}$$

(CONST) Similar to the (*var*) case when the variable is not substituted.

(SUB) Given the following typing derivation using (*sub*):

$$\frac{\Gamma_1, x : \tau_s, \Gamma_2 \mathbin{@} r' \vdash e : \tau}{\Gamma_1, x : \tau_s, \Gamma_2 \mathbin{@} r \vdash e : \tau} \quad (r' \leqslant r)$$

From the induction hypothesis, we have that $\Gamma_1, \Gamma, \Gamma_2 \mathbin{@} r' \circledast S \vdash e[x \leftarrow e_s] : \tau$. The condition on $\leqslant$ means that $r' \circledast S \leqslant r \circledast S$ and so we can apply the (*sub*) rule to obtain $\Gamma_1, \Gamma, \Gamma_2 \mathbin{@} r \circledast S \vdash e[x \leftarrow e_s] : \tau$.

(ABS) Given the following typing derivation using (*abs*):

$$\frac{\Gamma_1, x : \tau_s, \Gamma_2, y : \tau_1 \mathbin{@} r \wedge s \vdash e : \tau_2}{\Gamma_1, x : \tau_s, \Gamma_2 \mathbin{@} r \vdash \lambda y.e : \tau_1 \xrightarrow{s} \tau_2}$$

Assume w.l.o.g. that $x \neq y$. From the induction hypothesis, we have that:

$$\Gamma_1, \Gamma, \Gamma_2, y : \tau_1 \mathbin{@} r \circledast s \vdash e[x \leftarrow e_s] : \tau_2$$

Now using the fact that $\wedge = \circledast$, associativity and commutativity and (*abs*):

$$\frac{\dfrac{\dfrac{\Gamma_1, \Gamma, \Gamma_2, y : \tau_1 \mathbin{@} (r \wedge s) \circledast S \vdash e[x \leftarrow e_s] : \tau_2}{\Gamma_1, \Gamma, \Gamma_2, y : \tau_1 \mathbin{@} (r \circledast S) \wedge s \vdash e[x \leftarrow e_s] : \tau_2}}{\Gamma_1, \Gamma, \Gamma_2 \mathbin{@} r \circledast S \vdash \lambda y.(e[x \leftarrow e_s]) : \tau_1 \xrightarrow{s} \tau_2}}{\Gamma_1, \Gamma, \Gamma_2 \mathbin{@} r \circledast S \vdash (\lambda y.e)[x \leftarrow e_s] : \tau_1 \xrightarrow{s} \tau_2}$$

(APP) Given the following typing derivation using (*app*):

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

From the induction hypothesis, we have that:

$$\begin{array}{l} \Gamma_1, \Gamma, \Gamma_2 \mathbin{@} r \circledast S \vdash e_1[x \leftarrow e_s] : \tau_1 \xrightarrow{t} \tau_2 \\ \Gamma_1, \Gamma, \Gamma_2 \mathbin{@} s \circledast S \vdash e_2[x \leftarrow e_s] : \tau_1 \end{array} \quad (*)$$

Now using the (*app*) rule and the fact that $\oplus = \circledast$, associativity, commutativity and idempotence (note that all three properties are needed):

$$\frac{\dfrac{(*)}{\Gamma_1, \Gamma, \Gamma_2 \mathbin{@} (r \circledast S) \oplus ((s \circledast S) \circledast t) \vdash e_1[x \leftarrow e_s]\ e_2[x \leftarrow e_s] : \tau_2}}{\Gamma_1, \Gamma, \Gamma_2 \mathbin{@} (r \oplus (s \circledast t)) \circledast S \vdash (e_1\ e_2)[x \leftarrow e_s] : \tau_2}$$

(LET) Given the following typing derivation using (*let*):

$$\frac{\Gamma_1, x : \tau_s, \Gamma_2 \mathbin{@} r \vdash e_1 : \tau_1 \qquad \Gamma_1, x : \tau_s, \Gamma_2, y : \tau_1 \mathbin{@} s \vdash e_2 : \tau_2}{\Gamma_1, x : \tau_s, \Gamma_2 \mathbin{@} s \oplus (s \circledast r) \vdash \texttt{let}\ y = e_1\ \texttt{in}\ e_2 : \tau_2}$$

From the induction hypothesis, we have that:

$$\begin{array}{l} \Gamma_1, \Gamma, \Gamma_2 \mathbin{@} r \circledast S \vdash e_1[x \leftarrow e_s] : \tau_1 \\ \Gamma_1, \Gamma, \Gamma_2, y : \tau_1 \mathbin{@} s \circledast S \vdash e_2[x \leftarrow e_s] : \tau_2 \end{array} \quad (\dagger)$$

Now using the (*let*) rule and similarly to the (*app*) case:

$$\frac{\dfrac{(\dagger)}{\Gamma_1, \Gamma, \Gamma_2 \mathbin{@} (s \circledast S) \oplus ((s \circledast S) \circledast (r \circledast S)) \vdash \texttt{let}\ y = e_1[x \leftarrow e_s]\ \texttt{in}\ e_2[x \leftarrow e_s] : \tau_2}}{\Gamma_1, \Gamma, \Gamma_2 \mathbin{@} (s \oplus (s \circledast r)) \circledast S \vdash (\texttt{let}\ y = e_1\ \texttt{in}\ e_2)[x \leftarrow e_s] : \tau_2}$$

□

B.2 SUBSTITUTION FOR STRUCTURAL COEFFECTS

In order to prove that β-reduction and η-expansion preserve the type of an expression in Section 6.5.2, we required a multi-nary form of the substitution lemma (Lemma 40). This section provides the corresponding proof.

Lemma (Multi-nary substitution)**.** *Given an expression with multiple holes filled by variables* $x_{Ti} : \tau_{Ti}$ *with coeffects* s_k*:*

$$\Gamma \mathbin{@} \boldsymbol{r}\,[\, x_{T1} : \tau_{T1} \mathbin{@} \langle s_1 \rangle \mid \ldots \mid x_{Tk} : \tau_{Tk} \mathbin{@} \langle s_k \rangle\,] \vdash e_r : \tau_r$$

and a expressions e_{Ti} *with free-variable contexts* Γ_{Ti} *annotated with* $\mathbf{T_i}$*:*

$$\Gamma_1 \mathbin{@} \mathbf{T_1} \vdash e_{T1} : \tau_{T1} \quad \ldots \quad \Gamma_k \mathbin{@} \mathbf{T_k} \vdash e_{Tk} : \tau_{Tk}$$

then substituting the expressions e_{Ti} *for variables* x_{Ti} *results in an expression with a context where the original holes are filled by contexts* Γ_{Ti} *with coeffects* $s_i \circledast \mathbf{T_i}$*:*

$$\Gamma \mathbin{@} \boldsymbol{r}\,[\, \Gamma_{T1} \mathbin{@} s_1 \circledast \boldsymbol{T_1} \mid \ldots \mid \Gamma_{Tk} \mathbin{@} s_k \circledast \boldsymbol{T_k}\,] \vdash e_r[x_{T1} \leftarrow e_{T1}] \ldots [x_{Tk} \leftarrow e_{Tk}] : \tau_r$$

Proof. Assume that $\Gamma \mathbin{@} \mathbf{T_i} \vdash e_{Ti} : \tau_{Ti}$ and we are substituting terms e_{Ti} for variables x_{Ti}. Furthermore, we assume that are variables that are being substituted for actually appear in the original term (which means that k is at most the number of variables). Note that we use upper-case $\mathbf{T}$ to distinguish the coeffect of the expression that is being substituted into an expression. Using structural induction over $\vdash$:

SYNTAX-DRIVEN TYPING RULES

(VAR) Given the following derivation using (*var*):

$$\frac{}{y : \tau \mathbin{@} \langle \mathsf{use} \rangle \vdash y : \tau}$$

Here, the context contains exactly one variable and so $k = 1$. There are two cases depending on whether y is the (only substituted) variables x_{T1} or not:

- If $y = x_{T1}$ then also $\tau = \tau_{T1}$ and thus $y[x_{T1} \leftarrow e_{T_1}] = e_{T1}$. In this case, the context contains only a single hole $\Gamma \mathbin{@} \mathbf{r} = - \mathbin{@} -$. Using the assumption and the fact that use is a unit of $\circledast$:

$$\frac{\dfrac{\Gamma_{T1} \mathbin{@} \mathbf{T_i} \vdash y[x_{T1} \leftarrow e_{T1}] : \tau_r}{\Gamma \mathbin{@} \mathbf{r}\,[\, \Gamma_{T1} \mathbin{@} \mathbf{T_i}\,] \vdash y[x_{T1} \leftarrow e_{T1}] : \tau_r}}{\Gamma \mathbin{@} \mathbf{r}\,[\, \Gamma_{T1} \mathbin{@} \mathsf{use} \circledast \mathbf{T_i}\,] \vdash y[x_{T1} \leftarrow e_{T1}] : \tau_r}$$

- If $y \neq x_{T1}$ then there is no substitution that could be performed (y does not appear in the context) and so (trivially):

$$\Gamma \mathbin{@} \mathbf{r}\,[\,] \vdash y : \tau_r$$

(CONST) Similar to the (*var*) case when the variable is not substituted.

(APP) In the (*app*) rule, the context Γ is obtained as a tensor product Γ_1, Γ_2. Given Γ of length k, we assume that Γ_1 has length l and Γ_2 has length $k - l$. Now, given the following typing derivation using (*app*):

$$\frac{\begin{array}{c}\Gamma_1 \mathbin{@} \mathbf{s_1}[\, x_{T1} \mathbin{@} \langle s_1 \rangle \mid \ldots \mid x_{Tl} \mathbin{@} \langle s_l \rangle\,] \vdash e_1 : \tau_1 \xrightarrow{t} \tau_2 \\ \Gamma_2 \mathbin{@} \mathbf{s_2}[\, x_{Tl+1} \mathbin{@} \langle s_{l+1} \rangle \mid \ldots \mid x_{Tk} \mathbin{@} \langle s_k \rangle\,] \vdash e_2 : \tau_1\end{array}}{\begin{array}{l}\Gamma_1, \Gamma_2 \mathbin{@} \mathbf{s_1} \mathbin{+\!\!+} (t \circledast \mathbf{s_2})\ [\, x_{T1} \mathbin{@} \langle s_1 \rangle \mid \ldots \mid x_{Tl} \mathbin{@} \langle s_l \rangle, \\ \qquad\qquad x_{Tl+1} \mathbin{@} \langle t \circledast s_{l+1} \rangle \mid \ldots \mid x_{Tk} \mathbin{@} \langle t \circledast s_k \rangle\,] \vdash e_1\ e_2 : \tau_2\end{array}}$$

Here, we use $t \circledast s_2$ as a pointwise extension of $\wedge$ that is additionally applied to holes such that $t \wedge - = -$. That is, a hole in the context remains a hole and so the second part of the context is filled with $t \circledast s_i$ for all $i \in \{l+1 \ldots k\}$. Next, from the induction hypothesis, we have that:

$$\begin{array}{l} \Gamma_1 @ \mathbf{s_1}[\Gamma_{T1} @ s_1 \circledast \mathbf{T_1} \mid \ldots \mid \Gamma_{Tl} @ s_l \circledast \mathbf{T_l}] \vdash e_1\ [\ldots] : \tau_1 \xrightarrow{t} \tau_2 \\ \Gamma_2 @ \mathbf{s_2}[\Gamma_{Tl+1} @ s_{l+1} \circledast \mathbf{T_{l+1}} \mid \ldots \mid \Gamma_{Tk} @ s_k \circledast \mathbf{T_k}] \vdash e_2\ [\ldots] : \tau_1 \end{array} \quad (*)$$

Now using the (*app*) rule in the first step and associativity of $\circledast$ on the second part of the context in the second step:

$$\dfrac{(*)}{\begin{array}{l} \Gamma_1, \Gamma_2 @ \mathbf{s_1} \# (t \circledast \mathbf{s_2})\ [\Gamma_{T1} @ s_1 \circledast \mathbf{T_1} \mid \ldots \mid \Gamma_{Tl} @ s_l \circledast \mathbf{T_l}, \\ \quad \Gamma_{Tl+1} @ t \circledast (s_{l+1} \circledast \mathbf{T_{l+1}}) \mid \ldots \mid \Gamma_{Tk} @ t \circledast (s_k \circledast \mathbf{T_k})] \vdash (e_1\ e_2)\ [\ldots] : \tau_2 \end{array}}$$

$$\begin{array}{l} \Gamma_1, \Gamma_2 @ \mathbf{s_1} \# (t \circledast \mathbf{s_2})\ [\Gamma_{T1} @ s_1 \circledast \mathbf{T_1} \mid \ldots \mid \Gamma_{Tl} @ s_l \circledast \mathbf{T_l}, \\ \quad \Gamma_{Tl+1} @ (t \circledast s_{l+1}) \circledast \mathbf{T_{l+1}} \mid \ldots \mid \Gamma_{Tk} @ (t \circledast s_k) \circledast \mathbf{T_k}] \vdash (e_1\ e_2)\ [\ldots] : \tau_2 \end{array}$$

(ABS) Without the loss of generality, we can assume that the bound variable is not one of the variables being substituted. Thus, the last variable (and the corresponding coeffect) are not holes. The typing derivation using (*abs*) then looks as follows:

$$\dfrac{\Gamma, x{:}\tau_1 @ \mathbf{r} \# \langle s \rangle\ [x_{T1}{:}\tau_{T1} @ \langle s_1 \rangle \mid \ldots \mid x_{Tk}{:}\tau_{Tk} @ \langle s_k \rangle] \vdash e : \tau_2}{\Gamma @ \mathbf{r}\ [x_{T1}{:}\tau_{T1} @ \langle s_1 \rangle \mid \ldots \mid x_{Tk}{:}\tau_{Tk} @ \langle s_k \rangle] \vdash \lambda x.e : \tau_1 \xrightarrow{s} \tau_2}$$

From the induction hypothesis, we have that:

$$\Gamma, x{:}\tau_1 @ \mathbf{r} \# \langle s \rangle\ [\Gamma_{T1} @ s_1 \circledast \mathbf{T_1} \mid \ldots \mid \Gamma_{Tk} @ s_k \circledast \mathbf{T_k}] \vdash e\ [\ldots] : \tau_2$$

Because the last position in the vector of variables is an actual variable rather than a hole, we just need to apply the (*abs*) rule:

$$\dfrac{\Gamma, x{:}\tau_1 @ \mathbf{r} \# \langle s \rangle\ [\Gamma_{T1} @ s_1 \circledast \mathbf{T_1} \mid \ldots \mid \Gamma_{Tk} @ s_k \circledast \mathbf{T_k}] \vdash e\ [\ldots] : \tau_2}{\Gamma @ \mathbf{r}\ [\Gamma_{T1} @ s_1 \circledast \mathbf{T_1} \mid \ldots \mid \Gamma_{Tk} @ s_k \circledast \mathbf{T_k}] \vdash \lambda x.e\ [\ldots] : \tau_1 \xrightarrow{s} \tau_2}$$

(LET) In the structural coeffect calculus let-binding can be viewed as a syntactic sugar for abstraction/application and so the case follows from (*abs*) and (*app*).

Compared to the similar proof for the flat coeffect calculus, the proof for the structural system requires fewer properties of the coeffect algebra. In particular, we only needed associativity of $\circledast$ in the (*app*) rule the fact that use is a unit of $\circledast$ in (*var*).

STRUCTURAL RULES

(CONTR) In case of contraction, we can assume that the two variables to be contracted (in the assumption) are not the variables that are being substituted. However, the resulting variable (in the conclusion) can be one of the variables being substituted for.

- Assuming that $x \neq x_{Ti}$ for all i, the original derivation is:

$$\dfrac{\begin{array}{l} \Gamma_1, y{:}\tau_1, z{:}\tau_1, \Gamma_2 @ \mathbf{r} \# \langle s, t \rangle \# \mathbf{q} \\ \quad [x_{T1}{:}\tau_{T1} @ \langle s_1 \rangle \mid \ldots \mid x_{Tk}{:}\tau_{Tk} @ \langle s_k \rangle] \vdash e : \tau \end{array}}{\begin{array}{l} \Gamma_1, x{:}\tau_1, \Gamma_2 @ \mathbf{r} \# \langle s \oplus t \rangle \# \mathbf{q} \\ \quad [x_{T1}{:}\tau_{T1} @ \langle s_1 \rangle \mid \ldots \mid x_{Tk}{:}\tau_{Tk} @ \langle s_k \rangle] \vdash e[z, y \leftarrow x] : \tau \end{array}}$$

Applying (*contr*) to the induction hypothesis gives the required result:

$$\frac{\Gamma_1, y{:}\tau_1, z{:}\tau_1, \Gamma_2 \mathbin{@} \mathbf{r} \# \langle s, t\rangle \# \mathbf{q} \quad [\Gamma_{T1} \mathbin{@} s_1 \circledast \mathbf{T}_1 \mid \ldots \mid \Gamma_{Tk} \mathbin{@} s_k \circledast \mathbf{T}_k] \vdash e\,[\ldots] : \tau}{\Gamma_1, x{:}\tau_1, \Gamma_2 \mathbin{@} \mathbf{r} \# \langle s \oplus t\rangle \# \mathbf{q} \quad [\Gamma_{T1} \mathbin{@} s_1 \circledast \mathbf{T}_1 \mid \ldots \mid \Gamma_{Tk} \mathbin{@} s_k \circledast \mathbf{T}_k] \vdash e[z, y \leftarrow x][\ldots] : \tau}$$

- In the other case, $x = x_{Ti}$ for some i. The original typing is:

$$\frac{\Gamma_1, -, -, \Gamma_2 \mathbin{@} \mathbf{r} \# \langle -, -\rangle \# \mathbf{q}\ [x_{T1}{:}\tau_{T1} \mathbin{@} \langle s_1\rangle \mid \ldots \mid y{:}\tau_1 \mathbin{@} \langle s\rangle \mid z{:}\tau_1 \mathbin{@} \langle t\rangle \mid \ldots \mid x_{Tk}{:}\tau_{Tk} \mathbin{@} \langle s_k\rangle] \vdash e : \tau}{\Gamma_1, -, \Gamma_2 \mathbin{@} \mathbf{r} \# \langle - \rangle \# \mathbf{q}\ [x_{T1}{:}\tau_{T1} \mathbin{@} \langle s_1\rangle \mid \ldots \mid x{:}\tau_1 \mathbin{@} \langle s \oplus t\rangle \mid \ldots \mid x_{Tk}{:}\tau_{Tk} \mathbin{@} \langle s_k\rangle] \vdash e[z, y \leftarrow x] : \tau}$$

Applying (*contr*) to the induction hypothesis gives the following result:

$$\frac{\Gamma_1, -, -, \Gamma_2 \mathbin{@} \mathbf{r} \# \langle -, -\rangle \# \mathbf{q}\ [\Gamma_{T1} \mathbin{@} s_1 \circledast \mathbf{T}_1 \mid \ldots \mid \Gamma_{Ti} \mathbin{@} s \circledast \mathbf{T}_k \mid \Gamma_{Ti} \mathbin{@} t \circledast \mathbf{T}_k \mid \ldots \mid \Gamma_{Tk} \mathbin{@} s_k \circledast \mathbf{T}_k] \vdash e\,[\ldots] : \tau}{\Gamma_1, -, \Gamma_2 \mathbin{@} \mathbf{r} \# \langle - \rangle \# \mathbf{q}\ [\Gamma_{T1} \mathbin{@} s_1 \circledast \mathbf{T}_1 \mid \ldots \mid \Gamma_{Ti} \mathbin{@} (s \circledast \mathbf{T}_k) \oplus (t \circledast \mathbf{T}_k) \mid \ldots \mid \Gamma_{Tk} \mathbin{@} s_k \circledast \mathbf{T}_k] \vdash e\,[\ldots] : \tau}$$

Here the $\oplus$ operation represents a pointwise extension of $\oplus$. Thus for i^{th} substituted coeffect, we have $(s \circledast T_i) \oplus (t \circledast T_i)$. Using the distributivity law of structural coeffect algebra, we obtain the required structure: $(s \oplus t) \circledast T_i$.

(SUB) As in the (*contr*) case, in the (*sub*) case we distinguish two situations. If the subcoeffecting is applied to variable that is *not* being substituted for, then the case is easy (subcoeffecting does not interact with substitution), so we only consider the case when x is one of the variables being substituted for:

$$\frac{\Gamma_1, -, \Gamma_2 \mathbin{@} \mathbf{r} \# \langle - \rangle \# \mathbf{q}\ [x_{T1}{:}\tau_{T1} \mathbin{@} \langle s_1\rangle \mid \ldots \mid x{:}\tau_1 \mathbin{@} \langle s'\rangle \mid \ldots \mid x_{Tk}{:}\tau_{Tk} \mathbin{@} \langle s_k\rangle] \vdash e : \tau}{\Gamma_1, -, \Gamma_2 \mathbin{@} \mathbf{r} \# \langle - \rangle \# \mathbf{q}\ [x_{T1}{:}\tau_{T1} \mathbin{@} \langle s_1\rangle \mid \ldots \mid x{:}\tau_1 \mathbin{@} \langle s\rangle \mid \ldots \mid x_{Tk}{:}\tau_{Tk} \mathbin{@} \langle s_k\rangle] \vdash e : \tau}\ (s' \leqslant s)$$

From the induction hypothesis, we have the following:

$$\Gamma_1, -, \Gamma_2 \mathbin{@} \mathbf{r} \# \langle - \rangle \# \mathbf{q}\ [\Gamma_{T1} \mathbin{@} s_1 \circledast \mathbf{T}_1 \mid \ldots \mid \Gamma_{Ti} \mathbin{@} s' \circledast \mathbf{T}_k \mid \ldots \mid \Gamma_{Tk} \mathbin{@} s_k \circledast \mathbf{T}_k] \vdash e : \tau$$

To complete the case, we need to apply (*sub*) repeatedly on each of the variables in Γ_{Ti}. For i^{th} variable x_i, the coeffect annotation is $s' \circledast T_i$. Using the fact that combining coeffects with $\circledast$ preserves the ordering, we get that $(s' \circledast T_i) \leqslant (s \circledast T_i)$ and so the conditions of (*sub*) are satisfied.

(WEAK) We again need to consider whether the removed variable is one of the variables that are being substituted for. If this is not the case, the proof is easy, so we only look at the other case:

$$\frac{\Gamma \mathbin{@} \mathbf{r}\ [x_{T1}{:}\tau_{T1} \mathbin{@} \langle s_1\rangle \mid \ldots \mid x_{Tk}{:}\tau_{Tk} \mathbin{@} \langle s_k\rangle] \vdash e : \tau}{\Gamma, - \mathbin{@} \mathbf{r} \# \langle - \rangle\ [x_{T1}{:}\tau_{T1} \mathbin{@} \langle s_1\rangle \mid \ldots \mid x_{Tk}{:}\tau_{Tk} \mathbin{@} \langle s_k\rangle \mid x{:}\tau_1 \mathbin{@} \mathsf{ign}] \vdash e : \tau}$$

Now, we use the induction hypothesis, apply the (*weak*) rule and use properties of the structural coeffect algebra:

$$\dfrac{\Gamma \mathbin{@} \mathbf{r}\ [\,\Gamma_{T1} \mathbin{@} s_1 \circledast \mathbf{T}_1 \mid \ldots \mid \Gamma_{Tk-1} \mathbin{@} s_{k-1} \circledast \mathbf{T}_{k-1}\,] \vdash e\ [\ldots] : \tau}{(\textit{sub})\ \dfrac{\Gamma, - \mathbin{@} \mathbf{r} \# \langle - \rangle\ [\,\Gamma_{T1} \mathbin{@} s_1 \circledast \mathbf{T}_1 \mid \ldots \mid \Gamma_{Tk-1} \mathbin{@} s_{k-1} \circledast \mathbf{T}_{k-1} \mid x{:}\tau_1 \mathbin{@} \mathsf{ign}\,] \vdash e\ [\ldots] : \tau}{\Gamma, - \mathbin{@} \mathbf{r} \# \langle - \rangle\ [\,\Gamma_{T1} \mathbin{@} s_1 \circledast \mathbf{T}_1 \mid \ldots \mid \Gamma_{Tk-1} \mathbin{@} s_{k-1} \circledast \mathbf{T}_{k-1} \mid x{:}\tau_1 \mathbin{@} \mathsf{ign} \circledast \mathbf{T}_k\,] \vdash e\ [\ldots] : \tau}}$$

The derivation first applies the standard (*weak*) rule and then uses subcoeffecting rule and the property $\mathsf{ign} \leqslant (\mathsf{ign} \circledast r)$ to obtain conclusion of the required form.

(EXCH) In the (*exch*) case, the property follows directly from the induction hypothesis. The required conclusion is obtained by applying (*exch*) repeatedly (as we now need to exchange not just two individual variables, but two contexts, possibly containing multiple variables).

□