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, application running on a phone can access the GPS sensor, while application running in the cloud can access GPU computing resources. Both diversity and richness can only be expected to increase with trends such as the internet of things.

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

We start with a detailed discussion of the motivation for the thesis and an overview of our methodology (Chapter 1). Next, we discuss previous programming langauge 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 representative, but they illustrate various kinds of motivations well.

PARALLEL PROGRAMMING. The rise of multi-core CPUs is a clear example of an external development influencing programming language research. As multi-core and multi-processor systems became ubiquitous, languages had to provide better abstractions for parallel programming. This led to the industrial popularity of *immutable* data structures (and functional programming in general), software transactional memory [46], data-parallelism and also asynchronous computing [106].

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

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

Technologies like LINQ [66] make querying data significantly easier. But perhaps because accessing data became important more gradually, it was not easy to see that SQL queries, embedded as parameterized strings[2], are a poor solution *before* better approaches were developed.

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

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

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

[3] This is difficult to explain in writing and so the reader is encouraged to watch a video showing type providers for the WorldBank and CSV data sources [83].

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

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

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 [66] let developers write code in a host language like C# and then cross-compile it to multiple targets (including SQL, OpenCL or JavaScript [62]). 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 capabilities such as GPS sensor, social updates or battery status. We would like to know which of the capabilities are required and which are optional (i. e. enhance the user experience, but there is a fallback strategy). Equally, on the server-side, we might have access to different database tables, depending on the role of the user.

```
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

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

The following sub-sections explore 4 examples in more detail. The examples are chosen to demonstrate two distinct forms of contexts that are studied in this thesis – first two are related to the program environment and the latter two are associated with individual variables of the program.

1.1.1 *Context awareness #1: Platform versioning*

The diversity across devices means that developers need to target an increasing number of platforms and possibly also multiple versions of each platform. For Android, there is a number called API level [42] which "uniquely identifies the framework API revision offered by a version of the Android platform". Most changes in the libraries (but not all) are additive.

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

For example, the Figure 1 shows an excerpt from the Http module in the F# Data library[4]. The example uses conditional compilation to target multiple versions of the .NET framework. Such code is difficult to write – to see whether a change is correct, it had to be recompiled for all combinations of pre-processor flags – and maintaining the code is equally hard. The above example could be refactored and the .NET API could be cleaner, but the fundamental issue remains. If the language does not understand the context (here, the different platforms and platform versions), it cannot provide any static guarantees about the code.

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

As an alternative to conditional compilation, developers can use dynamic loading. For example, on Android, programs can access API from higher level platform dynamically using techniques like reflection and writing wrappers. This is even more error prone. As noted in an article[5] introducing the technique "Remember the mantra: if you haven't tried it, it doesn't work." Again, it would be reasonable to expect that statically-typed languages can provide a better solution.

1.1.2 *Context awareness #2: System capabilities*

Another example related to the previous one is when libraries use meta-programming techniques, such as LINQ [66, 22] or F# quotations [103], 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 targets multiple runtimes that have limited execution capabilities.

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

```
var db = new NorthwindDataContext();

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

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

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

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

When cross-compiling to a limited environment, we cannot always support the full source language. The example with LINQ and SQL demonstrates the importance of this problem. As of March 2014, Google search returns 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.

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

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

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 variables that are in some ways special. Such metadata can determine how the variables can be used. Here, name comes from the user input. This information about the value should be propagated to query. The SqlCommand object should then require arguments that can not directly contain user input (in an unchecked form).

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

In the security context, such marking of values (but at run-time) is called *tainting* [45], 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 data-flow programming examples used later in the thesis.

Consider a simple programming language with arrays where n^{th} element of an array arr is accessed using arr[n]. We focus on writing stencil computations (such as image blurring, Conway's game of life or convolution) where all arrays are of the same size and the system provides a *cursor* pointing to a current location in the stencil. We assume that the keyword cursor returns the current location in the stencil.

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

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

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

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

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 [78]. 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.

1.2.1 *Context-aware languages in action*

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

We assume that the language supports cross-compilation and splits the single program into three components: one for the server-side and two for the client-side, for iPhone and Windows platforms, respectively. The cross-compilation could be done in a way similar to Links [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

```
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

location and performs a call to the remote server and iPhoneMain, which constructs the user-interface. For Windows, we would also need fetchLocalNews, but this time with windowsMain. When using a language that can be compiled for all of the platforms, we would need a number of #if blocks to delimit the platform-specific parts.

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

1.2.2 *Understanding context with types*

The approach advocated in this thesis is to track information about context requirements using the type system. To make this practical, the system should also provide at least a partial support for automatic type inference, as the information about context requirements makes the types more complex. An inspiring example might be the F# support for units of measure [54] – the user has to explicitly annotate constants, but the rest of the information is inferred automatically.

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

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

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

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

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

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

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

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

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

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

- The fetchNews function requires database access and so it can only run on the server-side (or on a thick client with 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 in from `fetchLocalNews`, this function also requires GPS (the requirement is propagated automatically).

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

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

In this scenario, the main benefit of using an integrated context-aware language would be the ability to design appropriate abstractions using standard mechanisms of the language. For cross-compilation, we can structure code using functions, rather than relying on #if directives. Similarly, the splitting between client-side, server-side and shared code can be done using ordinary functions and modules (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# [109]:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- In Chapter 2, we give an overview of related work. Most importantly, we show that the idea of context-aware computations can be naturally approached from a number of directions developed recently in 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 data-flow, liveness analysis, distributed programming and Haskell's type classes). Our novel presentation reveals their similarity.

- In Chapter **??** and Chapter **??**, 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 **??** 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 **??**.

- In Chapter **??**, we use the translation as a basis for a prototype implementation of three simple context-aware programming languages. Our implementation serves as an empirical evidence showing how coeffects simplify programming with context and provide additional safety guarantees. We present the implementation in the format of an *interactive essay* (http://tomasp.net/coeffects) that encourages active reading and exploration of various aspects of the theory.

- Related work is presented in Chapter 2 and, together with further work, throughout the thesis. Two important directions deserve furhter exploration. In Chapter **??**, 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 (using *coeffects* or not). This greatly reduces the number of distinct concepts that software developers need to keep in mind of when building applications for the rich and diverse execution environments of the future.

2 PATHWAYS TO COEFFECTS

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

We start (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). Finally we consider logically inspired type systems that are closely related to our structural coeffects (Section 2.4).

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

2.1 THROUGH 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 [61], which provides an interesting point in the design space. We then discuss ambiguity that is inherent in variable binding and, more generally, context-aware programming languages, and a way of resolving it through *type-directed semantics*.

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 *requirements* can be satisfied in two standard ways that are captured by *dynamic* and *static* (or lexical) variable binding. Consider the following simple program:

```
let f =
    let x = 10 in
    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 langauge 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 last value that has been added to the environment 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 requirements 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 statis binding by default, but resort to dynamic binding if a variable is not available in the lexical scope. One such system is discussed in the next section.

Languages with static scoping generally check that all variable requirements are satisfied. Attempt to access a variable that is not available in scope will give a compile error. In contrast, languages with dynamic variable binding typically do not perform compile-time checks – if a program attempts to access a variable that is not available in the environment, a runtime error occurs. This is possibly artifact of their implementation – implementing static binding *without* checking would be cumbersome and implementing checking for dynamic binding is requires a more sophisticated type system (Section 2.2.2).

2.1.2 *Implicit parameter binding*

Haskell uses static binding for ordinary variables, but it additionally provides a feature named *implicit parameters* [61] 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 the Section 2.1.1. We replace a variable x with an implicit parameter ?x. On the left, the implicit parameter is defined both at declaration-site and at the call-site. On the right, the implicit parameter is available only at the call-site:

```
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 implicit parameters can be summarized as "*static binding when possible, dynamic binding when needed*". If an implicit parameter is available in 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 the function f has no delayed context requirements and the expression evalautes to 10. In the example on the right, the context requirement ?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. The type of the function f on the left is `int` → `int`, while the type of the f function on the right is $\{?x : \texttt{int}\} \Rightarrow \texttt{int} \rightarrow \texttt{int}$. 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 requirements too (Section 3.2.1).

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

In this thesis, we consider different notions of context, using implicit parameters as just one of several motivating examples. However, implicit parameters are a valuable example, because they clearly illustrate the ambiguity inherent in context-aware programs – the context requirements of a function can be satisfied using the context available at declaration-site or using the context available at the call-site.

We also aim to find a description of context-aware languages that does not make ad-hoc decisions about how context requirements are split between the declaration-site and the call-site. While Haskell's solution for implicit parameters might be the most reasonable one, different notions of context might 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 the type derivation. Typically, the choice is 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 shows a brief example using the F# language [109], before revisiting the implicit parameters example.

Consider an F# lambda expression ($\lambda x \to x.\mathsf{Length}$), 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 $\mathsf{string} \to \mathsf{int}$. In the second case, the list contains two arrays (empty array and array containing one hundred 0 values) and so the type of the lambda function is $\mathsf{int}[\,] \to \mathsf{int}$. The important points about the example are:

- The semantics of the function ($\lambda x \to x.\mathsf{Length}$) depends on its type. For arrays, it is compiled using a special IL instruction, while for strings, it calls a property getter.
- 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; in some cases, the compiler requires explicit typing annotation).

The function $\lambda y \to ?x + y$ in Haskell also has multiple possible typing derivations and its semantics varies depending on the type. If the lexical scope contains a binding for ?x, the function type is $\mathtt{int} \to \mathtt{int}$ and it captures the value from the lexical scope. Otherwise, the type of the function is $\{?x : \mathtt{int}\} \Rightarrow \mathtt{int} \to \mathtt{int}$ 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 (or fails, if type is not uniquely determined). 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 algorithm that chooses the preferred unique derivation.

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 how context requirements 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 (in case of implicit parameters, mimicking the behaviour of Haskell).

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

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

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

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

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

Figure 3: Simple effect system

2.2 THROUGH TYPE AND EFFECT SYSTEMS

Introduced by Gifford and Lucassen [39, 63], 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 [107], communication in message-passing systems [52] and atomicity in concurrent applications [35].

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. Filinsky [33] refers to this approach as *descriptive*[2].

2.2.1 *Simple effect system.*

The structure of a simple effect system[3] 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 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] In contrast to *prescriptive* effect systems that implement computational effects in a pure language – such as monads in Haskell.

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

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

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

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

$$(\mathit{app})\ \frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \xrightarrow{r} \tau_2, s \\ \Gamma \vdash e_2 : \tau_1, t\end{array}}{\Gamma \vdash e_1\ e_2 : \tau_2, r \cup s \cup t}$$

Figure 4: Simple coeffect system

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 $\mathtt{access}\ e$ that obtains the resource specified by the expression e. Most of the typing rules correspond to those of effect systems. Variable access (*var*) has no context requirements, application (*app*) combines context requirements of the two sub-expressions and latent context-requirements of the function. The (*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 time-zone, 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[4]. We discuss this system in detail in Section 3.2.1.

2.3 THROUGH LANGUAGE SEMANTICS

Another pathway to coeffects leads through the semantics of effectful and context-dependent computations. In a pioneering work, Moggi [68] showed that effects (including partiality, exceptions, non-determinism and I/O) can be modelled using the category theoretic notion of *monad*.

[4] 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.

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)* $M : \mathcal{C} \to \mathcal{C}$
- 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* $g : \beta \to M\gamma$*:*

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

Without providing much details, we note that well known examples of monads include the partiality monad ($M\alpha = \alpha + \bot$) also corresponding to the Maybe type in Haskell, list monad ($M\tau = 1 + (\tau \times M\tau)$) and other. 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"* [68, 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"* [68, 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 **??**). 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[5].

LANGUAGE SEMANTICS. When using monads to provide semantics of a language, we do not need to extend the language in any way – we assume that the language already contains the effectful primitives (such as the assignment operator $x \leftarrow e$ or other). A judgement of the form $x\!:\!\tau_1 \vdash e : \tau_2$ 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:

[5] 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.

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

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 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. This means that 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 actual effects) from a value and $\mathtt{let}_M$ is used to sequence effectful computations. In the semantics, monads are not needed to interpret variable access and application, they are only used in the semantics of additional (monadic) constructs:

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

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 [127]. 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 $\tau_1 \rightarrow M^r\tau_2$. Similarly, the semantics of a judgement $x : \tau_1 \vdash e : \tau_2, r$ can be given as a function $\tau_1 \rightarrow M^r\tau_2$. The precise nature of the family of monads has been later called *indexed monads* by Tate [108] and further developed by Atkey [7] in his work on *parameterized monads* and Katsumata [53].

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 data-flow computations [116] and more generally, notions of *context-dependent computations* [115]. 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. [4] discuss the relation between comonads and *containers* [3] 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[6]. In functional programming, Kieburtz [56] 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 [12], linking programming languages derived from modal logics to comonads. One such language has been reconstructed by Pfenning and Davies [88]. Nanevski et al. extend this work to Contextual Modal Type Theory (CMTT) [71], 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)* $\mathsf{C} : \mathcal{C} \rightarrow \mathcal{C}$
- counit *is a mapping* $\mathsf{C}\,\alpha \rightarrow \alpha$
- cobind *is a mapping* $(\mathsf{C}\,\alpha \rightarrow \beta) \rightarrow (\mathsf{C}\,\alpha \rightarrow \mathsf{C}\,\beta)$

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

[6] The structure of a computational comonad has been also used by the author of this thesis to abstract evaluation order of monadic computations [82].

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

The definition is similar to a monad with "reversed arrows". Intuitively, the counit operation extracts a value α from a value that carries additional context $C\alpha$. The cobind operation turns a context-dependent function $C\alpha \to \beta$ into a function that takes a value with context, applies the context-dependent function to value(s) in the context and then propagates the context. The next section makes this intuitive definition more concrete. More detailed discussion about comonads can be found in Orchard's PhD thesis [76].

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:

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

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

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

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

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

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

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

For example, the judgement $x : \tau \vdash \texttt{prev}\ (\texttt{prev}\ x) : \tau$ 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 *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 **??**.

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. [88, 71]. 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[7].

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 distributed programming language ML5 by Murphy et al. [69, 70], 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.

[7] In fact, Pfenning et al. never mention comonads explicitly. This is done in later work by Gabbay et al. [37], 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 **??**.

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 an type system that tracks the context requirements more precisely (by annotating the types) and we add indexing to comonads and link the two by giving a formal semantics. The 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. [37], 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 THROUGH 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 are simply added to the variable context (as a products), but we will later look at contextual properties that affect how variables are represented. More importantly, *structural coeffects* link additional information to individual variables in the context, rather than the context as a whole.

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

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

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

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 [128] restrict the use of variables in the language. Most famously linear type systems introduced by Wadler [125] 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 [74], 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 [75] (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:

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

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

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

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

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

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 can be used exactly once. In bunched typing, variables may still be duplicated, but only using the ";" separator. The type system can be interpreted as specifying whether a variable may be shared between the body of a function and the context where a function is declared.

The system introduces two distinct function types $\tau_1 \rightarrow \tau_2$ 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 [74].

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 **??**, we could define contexts as follows:

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

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 $(x\!:\!\tau_1) @ \mathsf{used}, (y\!:\!\tau_2, z\!:\!\tau_3) @ \mathsf{unused}$.

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 [98, 62] (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 [49]. The COP paradigm has been later implemented by programming language features. Costanza [26] develops a domain-specific LISP-like language ContextL and Bardram [8] 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 [10, 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.

CHAPTER STRUCTURE AND CONTRIBUTIONS

- We start with a characterization of contextual properties. The 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, data-flow, 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). The fact that we find a common structure in all systems presented here lets us develop unified coeffect calculi in the upcoming three chapters.
- In addition, the coeffect systems for tracking the number of accessed past values in data-flow languages (Sections 3.2.4 and 3.3.3) presents novel results and can be used to optimize data-flow programs.

As mentioned, this chapter may appear as a collection of disconnected examples[2]. But at the end, we will see that they share a common pattern.

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 envionment, or *input impurity*.

[1] A research programme, as introduced by Lakatos [59], 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!

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

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

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 $\Gamma @ \sigma \vdash e : \tau$, associating the context requirements with Γ. Thus, we extend the traditional notion of free-variable context Γ with richer notions of context. This notation emphasizes the right intuition, but there are more important differences between effects and coeffects.

3.1.1 *Effectful lambda abstraction*

The difference between effects and coeffects becomes apparent when we consider lambda abstraction. The typical lambda abstraction rule for effect systems looks as (*effect*) in Figure 8. Wadler and Thiemann [127] 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 [108] 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. In standard λ-calculus, variable access can be seen as a primitive operation that accesses the context. The variable access expression introduces a context requirement – the expression x is typeable only in a context that contains $x : \tau$ for some type τ.

The standard lambda abstraction (*pure*), 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 requirements can be tracked as part of the typing assumption, for example, say we have an expression e that requires GPS coordinates and the current time. The 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 [99], 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 a 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 details in Section 3.2.2. The important point here is that in flat coeffect systems, contextual requirements are *split* between the call site and declaration site. Furthermore, there is no syntactic structure that determines how the requirements are split. In the case of distributed programming, the resources can be freely associated with either of the two sites.

STRUCTURAL COEFFECTS. On the one hand, variable context provides a *fine-grained tracking* mechanism of how context (variables) are used. On the other hand, flat coeffects let us track *additional information* about the context. The purpose of *structural coeffects* is to reconcile the two and to provide a way for fine-grained tracking of additional information 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 that may be accessed relatively to the current cursor. 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 $\mathtt{cursor}$ as a language primitive. The coeffect annotations will be $(0, 0)$ for x and $(-1, 1)$ for y, denoting that we access only the current value in x, but we need access to both left and right neighbours in the y array. In order to unify the flat and structural notions, we attach this information as a *vector* of annotations associated with a *vector* of variable and write: $x : \mathsf{Arr},\ y : \mathsf{Arr} @ \langle (0,0), (-1,1) \rangle$. The unification is discussed in Chapter **??**.

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: $\lambda y.x[\mathtt{cursor}] = y[\mathtt{cursor} - 1] + y[\mathtt{cursor} + 1]$. Here, the declaration site contains x and needs to provide access at least within a range $(0,0)$. The call site provides a value for y, which needs to be accessible at least within $(-1,1)$. In this way, structural coeffects remove the non-determinism arising from the splitting of requirements in flat coeffect systems.

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

3.1.3 *Scalars and vectors*

The λ-calculus is asymmetric. It maps a context with *multiple* variables to a *single* result. An expression with n free variables of types τ_i can be modelled by a function $\tau_1 \times \ldots \times \tau_n \rightarrow \tau$ with a product on the left, but a single value on the right. 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 coeffect annotation per each variable. Thus, the annotation consists of multiple values – one belonging to each variable. To distinguish between the overall annotation and individual (per-variable) annotations, we call the overall coeffect a *vector* consisting of *scalar* coeffects. This asymmetry also explains why coeffect systems are not trivially dual to effect systems.

It is useful to clarify how vectors are used in this thesis. Suppose we have a set $\mathcal{C}$ of *scalars* 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[3]. 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 Chapter **??** and also discuss how our use relates to containers of Abbott, Altenkirch and Ghani [3].

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

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

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

3.2 FLAT COEFFECT SYSTEMS

In flat coeffect systems, the additional contextual information are 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 data-flow (Section 3.2.4).

For three of the examples, we present a type system and a simple semantics. 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 **??**.

3.2.1 *Implicit parameters and type classes*

Haskell provides two examples of flat coeffects – type class constraints and implicit parameter constraints [126, 61]. Both of the features introduce additional *constraints* on the context requiring that the environment provides certain operations for a type (type classes) or that it provides values for named implicit parameters. In the Haskell type system, constraints C are attached to the types of top-level declarations, such as let-bound functions. The Haskell notation $\Gamma \vdash e : C \Rightarrow \tau$ corresponds to our notation $\Gamma @ C \vdash e : \tau$.

In this section, we present a type system for implicit parameters in terms of the coeffect typing judgement. We briefly consider type classes, but do not give a full type system.

IMPLICIT PARAMETERS. 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 implicit parameters has an expression ?param to read a parameter and an expression[4] letdyn ?param $= e_1$ in e_2 that sets a parameter ?param to the value of e_1 and evaluates e_2 in a context containing ?param.

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

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

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

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

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

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

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

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

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

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

Figure 9: Coeffect rules for tracking implicit parameters

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

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

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

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

This is not a complete list of possible typings, but it demonstrates the options. The (*dynamic*) case requires the parameter ?width twice (this may be confusing when the caller provides two different values). In the (*mixed*) case, the nested function captures the ?width parameter available from the declaration site. 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 systems may introduce certain non-determinism in the typing. The following section shows how this looks in the type system for implicit parameters.

TYPE SYSTEM. Figure 9 shows a type system that tracks the set of expression's implicit parameters. The type system uses judgements of the form $\Gamma @ r \vdash e : \tau$ meaning that an expression e has a type τ in a free-variable context Γ with a set of implicit parameters specified by r. The annotations

r, s, t are finite partial functions mapping implicit parameter names to types, i. e. $r, s, t \subseteq \mathsf{Names} \mapsto \mathsf{Types}$. 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 requirements is (*param*). Accessing a parameter ?param of type τ requires it to be available in the context. The context may provide more (unused) implicit parameters thanks to the (*sub*) rule.

When we read the rules from the top to the bottom, application (*app*) and let binding (*let*) simply union the context requirements of the subexpressions. 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 ($\tau_1 \xrightarrow{s} \tau_2$). Finally, (*letdyn*) defines an implicit parameter and removes it 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 **??**:

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

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.

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 **??**.

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

The semantics is defined inductively over the typing derivation:

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

$$\llbracket \Gamma \,@\, r \vdash ?p : \sigma \rrbracket = \lambda(\_, f) \to f\ ?p \tag{\textit{param}}$$

$$\llbracket \Gamma \,@\, r \vdash e : \tau \rrbracket = \lambda(x, f) \to \llbracket \Gamma \,@\, r' \vdash e : \tau \rrbracket\ (x, f|_{r'}) \tag{\textit{sub}}$$

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

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

$$\begin{aligned} &\llbracket \Gamma \,@\, r \cup (s \setminus \{?p : \tau_1\}) \vdash \texttt{letdyn}\ ?p = e_1\ \texttt{in}\ e_2 : \tau_2 \rrbracket = \lambda(x, f) \to \\ &\qquad \texttt{let}\ v = \llbracket \Gamma \,@\, r \vdash e_1 : \tau_1 \rrbracket\ (x, f|_r) \\ &\qquad \texttt{in}\ \llbracket \Gamma \,@\, s \vdash e_2 : \tau_2 \rrbracket\ (x, f|_{s \setminus \{?p:\tau_1\}} \uplus \{?p \mapsto v\}) \end{aligned} \tag{\textit{letdyn}}$$

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

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

Figure 10: Semantics of a language with implicit parameters

$$\llbracket x_1 : \tau_1, \ldots, x_n : \tau_n \,@\, r \vdash e : \tau \rrbracket \ : \ (\tau_1 \times \ldots \times \tau_n) \times (r \to \sigma) \to \tau$$

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

$$\llbracket \tau_1 \xrightarrow{s} \tau_2 \rrbracket = \tau_1 \times (s \to \sigma) \to \tau_2$$

The definition of the semantics is shown in Figure 10. The (*var*) and (*param*) rules are simple – they project the appropriate variable and implicit parameter, respectively.

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

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

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).

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

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

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

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 **??**, 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 [126]. 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 $\mathsf{twoTimes}$ 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 $\mathsf{twoTimes}$ would look as follows:

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

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

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

SUMMARY. Implicit parameters are the simplest example of a system where function abstraction does not "delay" all impurities of the body. 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 introduces non-determinism in the type-inference, because the parameters can be split arbitrarily. However, as we show in the next section, this is not always the case. Semantically, lambda abstraction *merges* two parts of context (implicit parameter dictionaries) that are provided by the call site and declaration site.

3.2.2 *Distributed computing*

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

REBINDABLE RESOURCES. The need for parameters that support dynamic scoping also arises in distributed computing. To quote an example discussed by Bierman et al. [11]: *"Dynamic binding is required in various guises, for example when a marshalled value is received from the network, containing identifiers that must be rebound to local resources."*

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

A practical language that supports rebindable resources is for example Acute [99]. 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 recentNews = λ() →
  let db = access News in
  query db "SELECT * WHERE Date > %1" (access Clock)
```

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

// Searches database for a product; 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

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

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

CROSS-COMPILATION. A related issue with distributed programming is the need to target increasing number of diverse platforms. Modern applications often need to run on multiple platforms (iOS, Android, Windows or as JavaScript) or multiple versions of the same platform. Many programming languages are capable of targeting multiple different platforms. For example, functional languages that can be compiled to native code and JavaScript include, among others, F#, Haskell and OCaml [120].

Links [25], F# WebTools and WebSharper [105, 81], ML5 and QWeSST [69, 95] and Hop [62] go further and allow including code for multiple distinct platforms in a single source file. A single program is then automatically split and compiled to multiple target runtimes. This posses additional challenges – it is necessary to check where each part of the program can run and statically guarantee that it will be possible to compile code to the required target platform (safe *multi-targetting*).

We demonstrate the problem by looking at input validation. In applications that communicate over 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 implea.) Set-based type system for cross-compilation, inspired by Links [25]

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

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

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

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

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

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

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

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

mented 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 [81] 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 may seem similar to resource tracking (and thus to the tracking of implicit parameters), but it actually demonstrates a couple of new ideas that are important for flat coeffect systems. Unlike with implicit parameters, we will not present a specific existing system in this section – instead we briefly look at two examples that let us explore the range of possibilities.

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

Sub-coeffecting and application are trivially dual to the rules for implicit parameters. We just track supported environments using intersection as opposed to tracking 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 an call site environment and is executed there, possibly after it is marshalled and transferred to another machine.
- 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 [69]). 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 [69] 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 **??**.

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 semantics will be very similar.

Live variable analysis (LVA) [6] 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 [124] 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:

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

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

$$\begin{array}{lclclclclcl}
L \sqcup L & = & L & \quad & L \sqcup D & = & D & \quad & L \sqcap L & = & L & \quad & L \sqcap D & = & L \\
D \sqcup L & = & D & \quad & D \sqcup D & = & D & \quad & D \sqcap L & = & L & \quad & D \sqcap D & = & D
\end{array}$$

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

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

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

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

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

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

Figure 13: Liveness annotations with sequential and pointwise composition

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 of the system).

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

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

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

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

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

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

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

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

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

Figure 14: Coeffect rules for tracking whole-context liveness

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

The typing rules for tracking whole-context liveness are shown in Figure 14. The language now includes constants $c : \tau \in \Delta$. Accessing a constant (*const*) annotates the context as dead using D. This contrasts with variable access (*var*), which marks the context as live using L. A dead context (definitely not needed) can be treated as live context 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 the sequentially composed function (to evaluate e_2 followed by the function call). This is captured by *pointwise composition*.

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

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

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

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

$$(\lambda x.42)\ y \qquad (1)$$

$$\mathsf{twoTimes}\ 42 \qquad (2)$$

$$(\lambda x.x)\ 42 \qquad (3)$$

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

SEMANTICS. 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 **??**. For now, it suffices to define the semantics using two separate rules:

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

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. Sub-coeffecting is captured by two rules. A dead context can be treated as live using (*abs-1*); in other cases, the annotation is not changed (*abs-2*).

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

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

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

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 *Data-flow 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.

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

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

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

$$\llbracket \Gamma @ \mathsf{r} \vdash e : \tau \rrbracket = \lambda x \to \llbracket \Gamma @ \mathsf{r} \vdash e : \tau \rrbracket \; x \qquad (\textit{sub-2})$$

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

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

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

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

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

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

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

Lucid [123] is a declarative data-flow language designed by Wadge and Ashcroft. In Lucid, variables represent streams and programs are written as transformations over streams. A function application square(x) represents a stream of squares calculated from the stream of values x.

The data-flow approach has been successfully used in domains such as development of real-time embedded application where many *synchronous languages* [9] build on the data-flow paradigm. The following example is inspired by the Lustre [43] 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.

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

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

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

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

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

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

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

Formally, the constructs are defined as follows (writing x_n for n-th element of a stream x):

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

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

The semantics of Lucid-like languages can be captured using a number of mathematical structures. Wadge [122] originally defined a monadic semantics, while Uustalu and Vene later used comonads [114]. In Chapter **??**, we extend the latter approach. The present chapter presents a sketch of a concrete data-flow semantics defined directly on streams.

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

TYPE SYSTEM. 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; sub-coeffecting (*sub*) allows us to require more values than is actually needed. Primitive context-requirements are introduced in (*prev*), which increments the number of past values by one. Thus, for example, `prev` $($`prev` $x)$ requires 2 past values.

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

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

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

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

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

EXAMPLE. As with the liveness example, the application rule might require more explanation. The following example is somewhat arbitrary, but it demonstrates the rule well. We assume that counter is a stream of positive integers (starting from zero) and tick flips between 0 and 1. The full typing derivation is shown in Appendix **??**:

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

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

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

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

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

$$\begin{aligned}&[\![\Gamma @ n+1 \vdash \mathtt{prev}\ e : \tau]\!] = \lambda\langle\mathbf{v}_0,\ldots,\mathbf{v}_{n+1}\rangle \rightarrow\\ &\quad [\![\Gamma @ n \vdash e : \tau]\!]\ \langle\mathbf{v}_1,\ldots,\mathbf{v}_{n+1}\rangle\end{aligned} \tag{\textit{prev}}$$

$$\begin{aligned}&[\![\Gamma @ n \vdash e : \tau]\!] = \lambda\langle\mathbf{v}_0,\ldots,\mathbf{v}_n\rangle \rightarrow\\ &\quad [\![\Gamma @ n' \vdash e : \tau]\!]\ \langle\mathbf{v}_0,\ldots,\mathbf{v}_{n'}\rangle\end{aligned} \tag{\textit{sub}}$$

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

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

Figure 17: Semantics showing how past values are accessed in a data-flow language

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

Uustalu and Vene [115] model causal data-flow computations using a non-empty list $\mathsf{NeList}\ \tau = \tau \times (\mathsf{NeList}\ \tau + 1)$ over the input. A function $\tau_1 \rightarrow \tau_2$ is thus modelled as $\mathsf{NeList}\ \tau_1 \rightarrow \tau_2$. This model is difficult to implement efficiently, as it creates unbounded lists of past elements.

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

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

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

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 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 data-flow languages, for example to check initialization properties [24], but to our knowledge, not for checking the maximal number of required past values. Thus this section serves not just as an example, but also shows how coeffects can lead to novel results.

The most interesting point about the data-flow 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 **??**. Before doing that, we look at one more example and then explore the *structural* class of systems.

3.2.5 *Permissions and safe locking*

In the implicit parameters and data-flow examples, the context provides additional resources or values that may be accessed at runtime. However, 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 [34] as one, more advanced example. 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:

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

The declaration `newlock` creates a lock l protecting memory region ρ. We can than allocate mutable variables in that memory region (second line). An access to 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 [34] 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:

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

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 data-flow 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 per-variable.

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

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

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

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

- The fact that variables are now a vector means that we cannot freely reorder them. This guarantees that $x{:}\tau, y{:}\tau @ \langle \mathsf{L}, \mathsf{D} \rangle$ can not be confused with $y{:}\tau, x{:}\tau @ \langle \mathsf{L}, \mathsf{D} \rangle$. We need to define the type system in a way that is similar to 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 **??**, but the alternative is briefly explored in Section **??**.

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

The rules of the system are split into two groups. Figure 18 (a) shows the standard syntax-driven rules plus sub-coeffecting. In (*var*), the context contains just the single accessed variable, which is annotated as live. Unused variables can be introduced using weakening. A constant (*const*) is accessed in an empty context, which also carries no annotations. The sub-coeffecting rule (*sub*) uses a 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 $\mathbf{r} \times \langle \mathsf{s} \rangle$, 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 $\tau_1 \xrightarrow{\mathsf{t}} \tau_2$. The composition is defined as follows:

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

This definition is only for illustration and is revised in Chapter **??**. 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 $(\lambda x.e_2)\ e_1$).

a.) Ordinary, syntax-driven rules along with sub-coeffecting

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

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

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

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

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

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

b.) Structural rules for context manipulation

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

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

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

Figure 18: Structural coeffect liveness analysis

STRUCTURAL TYPING RULES. The structural typing rules are shown in Figure 18 (b). They mirror the rules know 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 $(\lambda x \to v)\ y$. This is similar to an example where flat liveness mistakenly marks the entire context as live. Despite the fact that the variable y is accessed (syntactically), it is not live – because the function that takes it as an argument always returns v.

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

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

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:

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

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

SEMANTICS. When defining the semantics of flat liveness calculus, we used an indexed form of the option type $1+\tau$ (which is 1 for dead contexts and τ for live contexts). In the semantics of expressions, the type 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:

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

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

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

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

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

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

As dictated by the semantics, the application again needs to "implement" dead code elimination (otherwise the type system would be unsound). When a.) Semantics of ordinary expressions

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

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

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

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

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

b.) Semantics of structural context manipulation

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

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

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

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

Figure 19: Semantics of structural liveness

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

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

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

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

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 we shall omit details that are the shared with the system in this section.

3.3.2 *Bounded variable use*

Liveness analysis checks whether a variable is used or unused. With structural coeffects, we can go further and track how many times is the variable accessed. Girard et al. [41] coined this idea as *bounded linear logic* and use it to restrict well-typed programs to polynomial-time algorithms. We first introduce the system in our, coeffect, style and then relate it with the original formulation.

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

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

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

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

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

a.) Ordinary, syntax-driven rules along with sub-coeffecting

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

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

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

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

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

b.) Structural rules for context manipulation

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

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

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

Figure 20: Structural coeffect bounded reuse analysis

Finally, the sub-coeffecting 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:

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

To type-check the application, the contexts of e_1 and e_2 need to contain disjoint variables. For this reason, we α-rename x to x' in the argument $(x + y)$ and later join x and x' using the contraction rule. Assuming $(x' + y)$ is checked in a context that marks x' and y as used once, the application rule yields a judgement that is simplified as follows:

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

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

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. [41] model their bounded linear logic in an (ordinary) linear logic where variables can be used at most once.

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

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

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

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

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

In the following, we use 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 $s = \langle s_1, \ldots, s_n \rangle$:

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

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

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

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

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

BOUNDED LINEAR LOGIC. The system presented in this section is based on the idea of bounded linear logic (BLL) [41], 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 annotation to the variable context as a whole. The requirements are attached to individual variables and so our context $\tau_1, \ldots, \tau_n @ \langle k_1, \ldots, k_n \rangle$ corresponds to a BLL assumption $!_{k_1} A_1, \ldots, !_{k_n} A_n$. Using the formulation of bounded logic (and omitting the terms), the weakening and contraction rules are written as follows:

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

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 **??**.

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

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. The main benefit of our approach is that it lets us unify flat and structural systems (Chapter **??**).

3.3.3 *Data-flow languages revisited*

When discussing data-flow 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 the same number of past values for all variables. However, data-flow can also be adapted to a structural system which keeps the number of required past values individually for each variable. Consider the following example:

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

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

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

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

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

Figure 21: Structural coeffect bounded reuse analysis

$$\texttt{let}\ \mathsf{offsetAdd} = \mathsf{left} + \texttt{prev}\ \mathsf{right}$$

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. Flat system is not able to capture this level-of-detail and simply requires 1 past values of both streams in the variable context.

Turning a flat data-flow system to a structural data-flow system is a change similar to the one between flat ans structural liveness. In case of liveness analysis, we included the flat system only as an illustration (it is not practically useful). For data-flow, the flat system is less precise, but still practically useful (simplicity may outweigh precision).

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

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

In flat data-flow, we identified sequential composition and pointwise composition as two primitive operations that were used in the (flat) application. In the structural system, these are used in (*app*) and (*contr*), respectively. Thus application combines coeffect annotations using $+$ and contraction using *max*. This contrasts with bounded reuse, which uses $*$ and $+$, respectively.

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

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

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:

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

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

SEMANTICS. To define the semantics of our structural data-flow language, we can use the same approach as when adapting flat liveness to structural liveness. Rather than wrapping the whole context in 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 as the structure used for bounded reuse. The only difference is that, given a variable annotated with r, we need $1 + r$ values. That is, we need the current value, followed by r past values:

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

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

$$\begin{aligned}
&\llbracket \Gamma\,@\,\langle (s_1+1), \ldots, (s_n+1)\rangle \vdash \mathtt{prev}\ e : \tau\rrbracket =\\
&\quad \lambda((x_{1,0}, \ldots, x_{1,s_1+1}), \ldots, (x_{n,0}, \ldots, x_{n,s_n+1})) \to\\
&\qquad \llbracket \Gamma\,@\,\langle s_1, \ldots, s_n\rangle \vdash e : \tau\rrbracket\\
&\qquad\quad ((x_{1,0}, \ldots, x_{1,s_1}), \ldots, (x_{n,0}, \ldots, x_{n,s_n}))
\end{aligned} \tag{\textit{prev}}$$

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

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

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

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

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

3.3.4 *Security, tainting and provenance*

Tainting is a mechanism where variables coming from potentially untrusted sources are marked (*tainted*) and the use of such variables is disallowed in contexts where untrusted input can cause security issues or other problems. Tainting can be done dynamically 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.

Static typed systems based on tainting have been use to prevent cross-site scripting attacks [118] and SQL injection attacks [45, 44]. In the latter case, we want to check that SQL commands cannot be directly constructed from, potentially dangerous, inputs provided by the user. Consider the type checking of the following expression in a context containing variables id and msg:

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

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

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

CORE DEPENDENCY CALCULUS. 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 credit card number) cannot be leaked to contexts with low secrecy (e. g. sent via an unsecured network channel). Volpano et al. [119] provide the first (provably) sound type system that guarantees non-inference and Sabelfeld et al. [94] surveys more recent work. Information flow checking has been also integrated (as a single-purpose extension) in the FlowCaml [100] language. Finally, Russo et al. and Swamy et al. [93, 102] 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 finite set of classes and an ordering. For example a set $\{\mathsf{L}, \mathsf{H}\}$ represents low and high secrecy, respectively with $\mathsf{L} \leqslant \mathsf{H}$ meaning that low security values can be treated as high security (but not the other way round).

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

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

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* [110], which detects which parts of programs can be partially evaluated (do not depend on user input) and *program slicing* [111] 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*):

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

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

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

3.4 BEYOND PASSIVE CONTEXTS

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

There is a number of systems that also capture contextual properties, but that make it possible to *change* the context – not just 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, for example, 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 [112] 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 = $\lambda$input $\rightarrow$
  letrgn $\rho$ in
  let x = $\text{ref}_\rho$ input in
  x := !x + 1; !x
```

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

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 = $\lambda$input $\rightarrow$
  letrgn $\rho$ in
  let x = $\text{ref}_\rho$ input in
  x := !x + 1; x
```

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

This is where systems with active context differ from systems with passive context. To type check such programs, we do not only need to know what context is required to call calculate (i. e. context on the left-hand side of $\vdash$). We also need to know what effects an expression has on the context when it evaluates and the current context meeds to be updated after a function call. This is an effectful property that would appear on the right-hand side of $\vdash$.

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 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 that specifies the context required before the call is performed and one that specifies how the context changes after the call (this could be either a *new* context or *update* to the original context). Thus the structure of the semantics would look as follows (with comonad to model coeffects on the left and 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) [36, 48]. The DSU systems have the ability to update programs at runtime without stopping them. For example, Proteus developed by Stoyle et al. [101] 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, data-flow languages such as Lucid, liveness analysis and also a number of security properties.

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

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

Part II

COEFFECT CALCULI

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

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

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

CHAPTER 4

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

CHAPTER 5

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

Part III

TOWARDS PRACTICAL COEFFECTS

TBD

BIBLIOGRAPHY

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[24] J.-L. Cola&#x00e7;o and M. Pouzet. Type-based initialization analysis of a synchronous dataflow language. *Int. J. Softw. Tools Technol. Transf.*, 6(3):245–255, Aug. 2004.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[76] D. Orchard. Programming contextual computations.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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