# 1 WHY CONTEXT-AWARE PROGRAMMING MATTERS

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

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

PARALLEL PROGRAMMING. The rise of multi-core CPUs is a clear example of an external development influencing programming language research. As multi-core and multi-processor systems became de-facto standard, languages had to provide better abstractions for parallel programming. This led to the industrial popularity of *immutable* data structures (and functional programming in general), software transactional memory [28], data-parallelism and also asynchronous computing [60].

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

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

Technologies like LINQ [38] make querying data significantly easier. But perhaps because accessing data became important more gradually, it was not easy to see that inline SQL is a poor solution *before* better approaches were developed.

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

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

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

[2] This is difficult to explain in writing and so the reader is encouraged to watch a video showing type providers for the WorldBank and CSV data sources [48].

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

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

## 1.1 WHY CONTEXT-AWARE PROGRAMMING MATTERS

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

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

- **System capabilities.** When writing code that is cross-compiled to multiple targets (e.g. SQL [38], OpenCL or JavaScript [36]) a part of the compilation (generating the SQL query) often occurs at runtime and developers have no guarantee that it will succeed until the program is executed.

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

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

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

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

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

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

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

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

### 1.1.1 *Context awareness #1: Platform versioning*

The diversity across devices means that developers need to target an increasing number of platforms and possibly also multiple versions of each platform. For Android, there is a number called API level [24] which "uniquely identifies the framework API revision offered by a version of the Android platform". Most changes in the libraries (but not all) are additive.

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

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

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

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

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

### 1.1.2 *Context awareness #2: System capabilities*

Another example related to the previous one is when libraries use metaprogramming techniques (such as LINQ [38] or F# quotations [58]) to translate code written in a subset of a host language to some other target language, such as SQL, OpenCL or JavaScript. This is an important technique, because it lets developers targets multiple heterogeneous runtimes that have limited execution capabilities.

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

```
var db = new NorthwindDataContext();

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

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

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

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

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

### 1.1.3 *Context awareness #3: Confidentiality and provenance*

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

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

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

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

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

The example demonstrates a more general property. Sometimes, it is desirable to track additional meta-data about variables that are in some ways special. Such meta-data can determine how the variables can be used. Here, name comes from the user input. This *provenance* information should be propagated to query. The SqlCommand object should then require arguments that can not directly contain user input (in an unchecked form). Such marking of values (but at run-time) is also called tainting [27].

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

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

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

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

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

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

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

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

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

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

## 1.2 TOWARDS CONTEXT-AWARE LANGUAGES

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

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

- **Flat coeffects** represent additional data, resources and meta-data that are available in the execution environment (regardless of how they are accessed in a program). Examples include resources such as GPS sensors and battery status (on a phone), databases (on the server), or framework version.

- **Structural coeffects** capture additional meta-data related to inputs. This can include provenance (source of the input value), usage information (how often is the value accessed and in what ways) or security information (whether it contain sensitive data or not).

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

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

### 1.2.1 *Context-aware language in action*

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

We assume that the language supports cross-compilation and splits the single program into three components: one for the server-side and two for the client-side, for iPhone and Windows platforms, respectively. The cross-compilation could be done in a way similar to Links [12], but we do not require explicit annotations specifying the target platform.

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

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

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

let iPhoneMain() =
  createCocoaListing(fetchLocalNews)

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

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

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

### 1.2.2 *Understanding context with types*

The approach advocated in this thesis is to track information about context requirements using the type system. To make this practical, the system needs to provide at least partial support for automatic type inference, as the information about context requirements makes the types more complex. An inspiring example might be the F# support for units of measure [32] – 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 $\tau_1$, produces a result of type $\tau_2$ and has additional context requirements specified by c. In the above example, the annotation c is simply a set of required resources or capabilities. However, a more complex structure could be used as well, for example, including the Android API level.

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

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

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

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

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

- The password variable is available in the context (we assume it has been declared earlier), but is marked as sensitive, which restricts how it can be used. In particular, we cannot return it as a result of a function that is called via a remote call (e. g. fetchNews) as that would leak sensitive data over an unsecured connection.
- The fetchNews function requires database access and so it can only run on the server-side (or on a thick client with local copy of the database, such as a desktop computer with an offline mode).
- The `gpsLocation` function accesses the GPS sensor and since we call it in from `fetchLocalNews`, this function also requires GPS (the requirement is propagated automatically).
- We can compile the program for two client-side platforms - the entry points are iPhoneMain and windowsMain and require Cocoa and Windows user-interface libraries, together with GPS and the ability to perform remote calls over the network.

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

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

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

## 1.3 THEORY OF CONTEXT DEPENDENCE

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## 1.4 THESIS OUTLINE

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

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

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

- In Chapter 2 and Chapter 3, we present the key novel contributions of this thesis. We develop the *flat* and *structural* calculi, show how they capture important contextual properties and develop their categorical semantics using a notion based on comonads. Chapter **??** links the two systems into a single formalism that is capable of capturing both flat and structural properties.
- Related work is presented in Chapter 2 and throughout the thesis, but one important direction deserves further exploration. In Chapter **??**, we look at a different approach to tracking contextual information that arises from modal logics. Finally, Chapter **??** discusses approaches for implementing the presented theory in main-stream programming languages and concludes.

# 2 PATHWAYS TO COEFFECTS

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

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

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

## 2.1 THROUGH TYPE AND EFFECT SYSTEMS

Introduced by Gifford and Lucassen [23, 37], 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 [61], communication in message-passing systems [31] and atomicity in concurrent applications [20].

Type and effect systems are usually specified judgements of the form $\Gamma \vdash e : \alpha, \sigma$, meaning that the expression $e$ has a type $\alpha$ in (free-variable) context $\Gamma$ and additionally may have effects described by $\sigma$. 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 [18] refers to this approach as *descriptive*[1].

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

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

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

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

Figure 3: Simple effect system

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

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

Figure 4: Simple effect system

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

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

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

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

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

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

### 2.2 THROUGH LANGUAGE SEMANTICS

Another pathway to coeffects leads through the semantics of effectful and context-dependent computations. In a pioneering work, Moggi [39] showed that effects (including partiality, exceptions, non-determinism and I/O) can be modelled uisng the category theoretic notion of *monad*.

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

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

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

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

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

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

#### 2.2.1 *Effectful languages and meta-languages*

Moggi uses monads to define two formal systems. In the first formal system, a monad is used to model the *language* itself. This means that the semantics of a language is given in terms of a one specific monad and the semantics can be used to reason about programs in that language. To quote *"When reasoning about programs one has only one monad, because the programming language is fixed, and the main aim is to prove properties of programs"* [39, 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"* [39, p. 5].

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

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

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

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

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

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

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

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

The types consist of primitive type ($\tau$), 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 $\alpha$ and computations $M\alpha$. The additional expression $\mathbf{return}_M$ is used to create a monadic computation (with no actual effects) from a value and $\mathbf{let}_M$ is used to sequence effectful computations. In the semantics, monads are not needed to interpret variable access and application, they are only used in the semantics of additional (monadic) constructs:

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

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

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

### 2.2.2 *Marriage of effects and monads*

The work on effect systems and monads both tackle the same problem – representing and tracking of computational effects. The two lines of research have been joined by Wadler and Thiemann [77]. This requires extending the categorical structure. A monadic computation $\alpha \to M\beta$ means that the computation has *some* effects while the judgement $\Gamma \vdash e : \alpha, \sigma$ specifies *what* effects the computation has.

To solve this mismatch, Wadler and Thiemann use a *family* of monads $M^{\sigma}\alpha$ with an annotation that specifies the effects that may be performed by the computation. In their system, an effectful function $\alpha \xrightarrow{\sigma} \beta$ is modelled as a pure function returning monadic computation $\alpha \to M^{\sigma}\beta$. Similarly, the semantics of a judgement $x : \alpha \vdash e : \beta, \sigma$ can be given as a function $\alpha \to M^{\sigma}\beta$. The precise nature of the family of monads has been later called *indexed monads* (e.g. by Tate [62]) and further developed by Atkey [4] in his work on *parameterized monads*.

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

### 2.2.3 *Context-dependent languages and meta-languages*

The theoretical parts of this thesis extend the work of Uustalu and Vene who use comonads to give the semantics of data-flow computations [68] and more generally, notions of *context-dependent computations* [67]. The computations discussed in the latter work include streams, arrays and containers – this is a more diverse set of examples, but they all mostly represent forms of collections. Ahman et al. [2] discuss the relation between comonads and *containers* in more details.

The utility of comonads has been explored by a number of authors before. Brookes and Geva [10] use *computational* comonads for intensional semantics[3]. In functional programming, Kieburtz [33] proposed to use comonads for stream programming, but also handling of I/O and interoperability.

Biermann and de Paiva used comonads to model the necessity modality $\Box$ in intuitionistic modal S4 [9], linking programming languages derived from modal logics to comonads. One such language has been reconstructed by Pfenning and Davies [49]. Nanevski et al. extend this work to Contextual Modal Type Theory (CMTT) [42], which again shows the importance of comonads for *context-dependent* computations.

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

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

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

[3] The structure of computational comonad has been also used by the author of this thesis to abstract evaluation order of monadic computations [47].

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

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

The definition is similar to monad with "reversed arrows". Intuitively, the counit operation extracts a value $\alpha$ 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 [45].

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

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

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

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

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

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

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

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

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

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

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

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

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

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

In languages inspired by modal logics, types can have the form $\Box\alpha$. In the work of Pfenning and Davies, this means a term that is provable with no assumptions. In distributed programming language ML5, Murphy et al. [40, 41] use the $\Box\alpha$ type to mean *mobile code*, that is code that can be evaluated at any node of a distributed system (the evaluation corresponds to the axiom $\Box\alpha \rightarrow \alpha$). Finally, Davies and Pfenning [15] consider staged computations and interpret $\Box\alpha$ as a type of (unevaluated) expressions of type $\alpha$.

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

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

- (*eval*) corresponds to $\mathsf{counit} : C^{\emptyset}\alpha \rightarrow \alpha$. It means that we can evaluate a closed (unevaluated) term and obtain a value. Note that the rule requires a specific context annotation. It is not possible to evaluate an open term.

[4] In fact, Pfenning and Davies [49, 42] never mention comonads explicitly. This is done in later work by Gabbay et al. [22], but the connection between the language and comonads is not as direct as in case of monadic or comonadic semantics covered in the last few pages.

- (*letbox*) corresponds to $\mathsf{cobind} : (C^{\Psi}\alpha \to \beta) \to C^{\Psi,\Phi}\alpha \to C^{\Phi}\beta$. It means that given a term which requires variable context $\Psi, \Phi$ (expression $e_1$) and a function that turns a term needing $\Psi$ into an evaluated value (expression $e_2$), we can construct a term that requires just $\Phi$.

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

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

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

The *meta-language* approach of Pfenning, Davies and Nanevski et al. is closely related to our work. Most importantly, Contextual Modal Type Theory (CMTT) uses indexed $\Box$ modality which seems to correspond to indexed comonads (in a similar way in which effect systems correspond to indexed monads). The relation between CMTT and comonads has been suggested by Gabbay et al. [22], but the meta-language employed by CMTT does not directly correspond to comonadic operations. For example, our $\mathsf{let\ box}$ typing rule from Figure 5 is not a primitive of CMTT and would correspond to $\mathsf{box}(\Psi, \mathsf{letbox}(e_1, x, e_2))$. Nevertheless, the indexing in CMTT provides a useful hint for adding indexing to the work of Uustalu and Vene.

## 2.3 THROUGH SUB-STRUCTURAL AND BUNCHED LOGICS

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

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

In this section, we look at type systems that reconsider $\Gamma$ in a number of ways. First of all, sub-structural type systems [78] restrict the use of variables in the language. Most famously linear type systems introduced by Wadler [75] can guarantee that variable is used exactly once. This has interesting implications for memory management and I/O.

In bunched typing developed by O'Hearn [43], the variable context is a tree formed by multiple different constructors (e.g. one that allows sharing

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

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

Figure 6: Exchange, weakening and contraction typing rules

and one that does not). Most importantly, bunched typing has contributed to the development of separation logic [44] (starting a fruitful line of research in software verification), but it is also interesting on its own.

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

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

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

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

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

The values I and 1 represent two kinds of "empty" contexts. More interestingly, non-empty variable contexts may be constructed using two distinct constructors – $\Gamma, \Gamma$ 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 $\Delta$ 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

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

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

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

be interpreted as specifying whether a variable may be shared between the body of a function and the context where a function is declared. The system introduces two distinct function types $\alpha \to \beta$ and $\alpha \mathrel{-\!\!*} \beta$ (corresponding to ";" and "," respectively). The key property is that only the first kind of functions can share variables with the context where a function is declared, while the second restricts such sharing. We do not attempt to give a detailed description here as it is not immediately to coeffects – for more information, refer to O'Hearn's introduction [43].

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

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

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

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

## 2.4 SUMMARY

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

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

# 3 CONTEXT-AWARE APPLICATIONS

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

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

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

## 3.1 STRUCTURE OF COEFFECT SYSTEMS

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

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

### 3.1.1 *Lambda abstraction*

The difference between effects and coeffects becomes apparent when we consider lambda abstraction. The typical lambda abstraction rule for effect systems looks as (*abs-eff*) in Figure 8. Wadler and Thiemann [77] explain how the effect analysis works as follows:

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

[1] Research programme, as introduced by Lakatos [34], is a network of scientists sharing the same basic assumptions and techniques.

$$\text{(abs-pure)}\ \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \rightarrow \tau_2 1} \qquad \text{(abs-eff)}\ \frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \,\&\, \sigma}{\Gamma \vdash \lambda x.e : \tau_1 \xrightarrow{\sigma} \tau_2 \,\&\, \emptyset}$$

Figure 8: Lambda abstraction for pure and effectful computations

This is the key property of *output impurity*. The effects are only produced when the function is evaluated and so the effects of the body are attached to the function. A recent work by Tate [62] 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 $\lambda$-calculus. This is well understood and we use it to demonstrate how contextual properties behave. Then we consider two notions of context introduced in this thesis – *flat coeffects* refer to overall properties of the environment and *structural coeffects* refer to properties attached to individual variables.

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

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

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

FLAT COEFFECTS. In Chapter 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 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 and current time. The context of such expression will be $\Gamma \,@\, \{\, \mathsf{gps}, \mathsf{time} \,\}$.

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

In the example discussed here, $\lambda x.e$ would require a context $\Gamma @ \{ \texttt{gps} \}$

Context requirements in distributed systems can be satisfied in a number of ways. When the resource is available locally,

In distributed systems, additional resources available at certain nodes (e. g. a database or GPS sensor) can be also viewed as a part of typing assumptions.

Similarly, resources in a distributed system can be satisfied by both the declaration site (by capturing a remote reference) and the call site (by resource rebinding) [54]. In this case, the resource requirements are split freely.

We discuss this example in detail in Section X.

STRUCTURAL COEFFECTS. The previously described notion of a *coeffect system* [**?** ] tracks contextual properties of entire programs, which is practically useful for implicit parameters or resources. However, for per-variable coeffects, the previous work provides only an approximation with limited practical usefulness (e. g. marking the whole context as live when just one variable in the context is live).

Furthermore, contextual requirements may relate to specific variables. For example, liveness or access patterns, such as the number of accesses in bounded linear logic or past values in causal dataflow.

For contextual properties associated with variables (e. g. liveness or dataflow), the splitting tracks free variables – requirements associated with a *bound* variable should be satisfied by the call site, while those associated with *free* variables should be provided by the declaration site.

lambda abstraction places requirements on both the *call-site* (latent requirements) and the *declaration-site* (immediate requirements),

We start with some background and finish with a brief overview of the literature leading to coeffects.

In the rest of the chapter, we look at instances of both. But before, we need to look at vectors.

### 3.1.3 *Scalars and vectors*

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

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

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

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

for the pointwise extension of $\leqslant$ on scalars. Finally, the associative operation $\times$ concatenates vectors.

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

## 3.2 flat coeffects (1.)

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

implicit parameters In Haskell, implicit parameters [35] are a special kind of variables that may behave as dynamically scoped. This means, if a function uses parameter ?p, then the caller of the function must define ?p and set its value. Implicit parameters can be used to parameterise a computation (involving a chain of function calls) without passing parameters explicitly as additional arguments of all involved functions. A simple language with implicit parameters has an expression ?p to read a parameter value and an expression[3] **letdyn** ?p $= e_1$ **in** $e_2$ that sets a parameter ?p to the value of $e_1$ and evaluates $e_2$ in a context containing ?p

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

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

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

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

```
(?width :: Int) ⇒ String → ((?size :: Int) ⇒ String → string)
```

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

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

This way of assigning type to format and calling it is not the only possible, though. We could also say that the outer function requires both of the im-

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

plicit parameters and the result is a (pure) function with no context requirements. This interaction between implicit parameters and lambda abstraction demonstrates one of the key aspects of coeffects and will be discussed later. Implicit parameters will also sever as one of our examples in Chapter Y.

TYPE CLASSES Implicit parameters are closely related to *type classes* [76]. In Haskell, type classes provide a principled form of ad-hoc polymorphism (overloading). When a code uses an overloaded operation (e.g. comparison or numeric operators) a constraint is placed on the context in which the operation is used. For example:

$$\mathsf{twoTimes} :: \mathsf{Num}\ \alpha \Rightarrow \alpha \rightarrow \alpha$$
$$\mathsf{twoTimes}\ x = x + x$$

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

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

REBINDABLE RESOURCES The need for parameters that do not strictly follow static scoping rules also arises in distributed computing. This problem has been addressed, for example, by Bierman et al. and Sewell et al. [8, 54]. To quote the first work: *"Dynamic binding is required in various guises, for example when a marshalled value is received from the network, containing identifiers that must be rebound to local resources."*

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

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

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

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

**distributed computing and multi-targetting** An increasing number of programming languages is capable of running across multiple different platforms or execution environments. Functional programming languages that can be compiled to JavaScript (to target web and mobile clients) include, among others, F#, Haskell and OCaml [71].

Links [12], F# libraries [59, 46], ML5 and QWeSST [40, 52] and Hop [36] go further and allow a single source program to be compiled to multiple target runtimes. This posses additional challenges – it is necessary to track where each part of computation runs and statically guarantee that it will be possible to compile code to the required target platform (safe *multi-targetting*).

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

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

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

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

In Links [12], functions can be annotated as client-side, server-side and database-side. F# WebTools [46] adds functions that support multiple targets (mixed-side). However, these are single-purpose language features and they are not extensible. For example, in modern mobile development it is also important to track minimal supported version of runtime[4].

Requirements on the execution environment can be viewed as contextual properties, but could be also presented as effects (use of some API required only in certain environment is a computational effect). We discuss the difference in Section X. Furthermore, the theoretical foundations of distributed languages like ML5 [40] suggest that a contextual treatment is more appropriate. We return to ML5 when discussing semantics in Section 2.2.3.

**safe locking** In the previous examples, the context provides additional values or functions that may be accessed at runtime. However, it may also track *permissions* to perform some operation. This is done in the type system for safe locking of Flanagan and Abadi [19].

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

```
newlock l : ρ in
let state  =  ref$_\rho$ 10 in
sync l (!state)
```

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

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

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

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

Data-flow languages provide a different example. Lucid [73] 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 $\mathit{square}(a)$ represents a stream of squares calculated from the stream of values $a$.

The data-flow approach has been successfully used in domains such as development of real-time embedded application where many *synchronous languages* [6] build on the data-flow paradigm. The following example is inspired by the Lustre [25] language and implements program to count the number of edges on a Boolean stream:

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

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

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

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

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

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

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

Figure 9: Selected coeffect rules for implicit parameters

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

To understand the nature of the context, we later look at the semantics of Lucid. This can be captured using a number of mathematical structures. Wadge [72] originally proposed to use monads, while Uustalu and Vene later used comonads [66].

## 3.3 FLAT COEFFECTS (2.)

### 3.3.1 *Implicit parameters and resources.*

Implicit parameters [35] are *dynamically-scoped* variables. They can be used to parameterize a computation without propagating arguments explicitly through a chain of calls and are part of the context in which expressions evaluate. As correctly expected [35], they can be modelled by comonads. Rebindable resources in distributed computations follow a similar pattern, but we discuss implicit parameters for simplicity.

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

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

Figure 9 shows a type-and-coeffect system tracking the set of an expression's implicit parameters. For simplicity here, all implicit parameters have type $\rho$.

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

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

The coeffect system infers multiple, i. e. non-principal, coeffects for functions. Different judgments are desirable depending on how a function is used. In the first case, both parameters have to be provided by the caller. In the second, both are available at declaration-site, but ?format may be rebound (precise meaning is provided by the monoidal structure on the product comonad in §**??**).

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

### 3.3.2 *Liveness analysis.*

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

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

A coeffect system in Fig. 10 detects whether all variables are dead ($C^{D}\Gamma$) or whether at least one variable is live ($C^{L}\Gamma$). Variable access (*var*) is annotated with $L$ and constant access with $D$. That is, if $c \in \mathbb{N}$ then $C^{D}\Gamma \vdash c : \mathsf{int}$. A dead context may be marked as live by letting $D \sqsubseteq L$ and adding subcoeffecting (§**??**).

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

An (*abs*) rule (not shown) compatible with the structure in Fig. 9 combines the context annotations using $\sqcap$. Thus, if the body uses some variables, both the function argument and the context of the declaration-site are marked as live.

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

### 3.3.3 *Efficient dataflow.*

Dataflow languages (e. g. [73]) declaratively describe computations over streams. In *causal* data flow, program may access past values – in this setting, a function $\tau_1 \to \tau_2$ becomes a function from a list of historical values $[\tau_1] \to \tau_2$. A coeffect system here tracks how many past values to cache.

Figure 11 annotates contexts with an integer specifying the maximum number of required past values. The current value is always present, so (*var*) is annotated with $0$. The expression **prev** $e$ gets the previous value of stream $e$ and requires one additional past value (*prev*); e. g. **prev** (**prev** $e$) requires 2 past values.

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

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

Figure 10: Selected coeffect rules for liveness analysis

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

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

Figure 11: Selected coeffect rules for causal data flow

the number of past values required by the function body (the body may use variables coming from the declaration-site as well as the argument).

The soundness follows from our categorical model (§**??**). Uustalu and Vene [67] model causal dataflow computations using a non-empty list comonad $\mathsf{NeList}\ \tau = \tau \times (\mathsf{NeList}\ \tau + 1)$. However, such model leads to (inefficient) unbounded lists of past elements. The above static analysis provides an approximation of the number of required past elements and so we use just fixed-length lists.

## 3.4 STRUCTURAL COEFFECTS (1.)

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

LIVENESS ANALYSIS *Live variable analysis* (LVA) [3] is a standard technique in compiler theory. It detects whether a free variable of an expression may be used by a program later (it is *live*) or whether it is definitely not needed (it is *dead*). As an optimization, compiler can remove bindings to dead variables as the result is never accessed. Wadler [74] describes the property of a variable that is dead as the *absence* of a variable.

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

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

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

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

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

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

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

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

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

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

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

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

TAINTING AND PROVENANCE Tainting is a mechanism where variables coming from potentially untrusted sources are marked (*tainted*) and the use of such variables is disallowed in contexts where untrusted input can cause security issues or other problems. Tainting can be done dynamically as a runtime mark (e.g. in the Perl language) or statically using a type system. Tainting can be viewed as a special case of *provenance tracking*, known from database systems [11], where values are annotated with more detailed information about their source.

Statically typed systems that based on tainting have been use to prevent cross-site scripting attacks [69] and a well known attack known as SQL injection [27, 26]. In the latter chase, we want to check that SQL commands cannot be directly constructed from, potentially dangerous, inputs provided by the user. Consider the type checking of the following expression in a context containing variables id and msg:

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

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

In runtime checking, all (string) values need to be wrapped in an object that stores Boolean flag (for tainting) or more complex data (for provenance).

In static checking, the information need to be associated with the variables in the variable context. We use tainting as a motivating example for *structural* coeffects in Section X.

SECURITY AND CORE DEPENDENCY CALCULUS The checking of tainting is a special case of checking of the *non-interference* property in *secure information flow*. Here, the aim is to guarantee that sensitive information (such as credit card number) cannot be leaked to contexts with low secrecy (e.g. sent via an unsecured network channel). Volpano et al. [70] provide the first (provably) sound type system that guarantees non-inference and Sabelfeld et al. [51] survey more recent work. The checking of information flows has been also integrated (as a single-purpose extension) in the FlowCaml [55] language. Finally, Russo et al. and Swamy et al. [50, 57] show that the properties can be checked using a monadic library.

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

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

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

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

Abadi et al. realized that there is a number of analyses similar to secure information flow and proposed to unify them using a single model called Dependency Core Calculus (DCC) [1]. It captures other cases where some information about expression relies on properties of variables in the context where it executes. The DCC captures, for example, *binding time analysis* [63], which detects which parts of programs can be partially evaluated (do not depend on user input) and *program slicing* [64] that identifies parts of programs that contribute to the output of an expression.

## 3.5 STRUCTURAL COEFFECTS

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

### 3.5.1 *Bounded reuse*

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

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

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

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

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

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

Figure 12: type and coeffect system for bounded reuse

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

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

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

BOUNDED LINEAR COEFFECTS. The system in Figure 12 extends the outlined idea into a simple calculus. Variable access (*var*) has a singleton context with a singleton coeffect vector $\langle 1\rangle$. Weakening (*weak*) extends the free-variable context with an unused variable and the coeffect with an associated scalar $0$. Explicit contraction (*contr*) and exchange (*exch*) rules manipulate variables in the context and modify the annotations accordingly – adding the number of uses in contraction and switching vector elements in exchange.

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

Application (*app*) describes call-by-name evaluation. Applying a function that uses its parameter $t$-times to an argument that uses variables in $\Gamma_2$ $S$-times means that, in total, the variables in $\Gamma_2$ will be used $(t * S)$-times. Recall that $t * S$ is a scalar multiplication of a vector. Meanwhile, the variables in $\Gamma_1$ are used just $R$-times when reducing the expression $e_1$ to a function value.

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

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

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

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

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

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

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

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

### 3.5.2 *Dataflow and data access*

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

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

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

Weakening, exchange, abstraction and sub-coeffecting are the same as in bounded linear coeffects, but the remaining rules differ. In Figure 13, accessed variables (*var*) are annotated with $0$ meaning that no past value is required (only the current one). The (*prev*) rule crates caching requirements – it increments the number of required values for all variables used in $e$ using scalar-vector addition.

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

In (*app*), the function requires $t$ past values of its parameter. This means $t$ past values of $e_2$ are needed which in turn requires $S$ past values of its free variables $\Gamma_2$. Thus, we need $t + S$ past values of $\Gamma_2$ to perform the call (e. g., we need $1 + S$ values to get 1 past value of the input $\sigma$, $2 + S$ values to get 2 past values of $\sigma$, *etc.*).

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

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

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

Figure 13: type and coeffect system for dataflow caching

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

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

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 $\mathbf{prev}\ (\mathbf{prev}\ y)$ requires two past values of $y$ and so the overall requirement is 3 past values:

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

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

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

### 3.5.3 *Implicit parameters*

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

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

Despite the differences, the type system in Figure 14 follows the same structure as the earlier two examples. Context requirements are created

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

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

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

Figure 14: Type and coeffect system for implicit parameters

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

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

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

EXAMPLE. Unlike structural coeffect systems, flat systems do not necessarily have principal coeffects. This arises from the (*abs*) rule which can freely split requirements between the function type and the declaring context. Consider a function $\lambda().?\mathsf{p}_1 + ?\mathsf{p}_2$. There are nine possible type and coeffect derivations, two of which are:

$$\begin{aligned} \emptyset \,@\, \{\} &\vdash (\ldots) : \mathsf{unit} \xrightarrow{\{?\mathsf{p}_1 : \mathbb{Z}, ?\mathsf{p}_2 : \mathbb{Z}\}} \mathbb{Z} \\ \emptyset \,@\, \{?\mathsf{p}_1 : \mathbb{Z}\} &\vdash (\ldots) : \mathsf{unit} \xrightarrow{\{?\mathsf{p}_2 : \mathbb{Z}\}} \mathbb{Z} \end{aligned}$$

In the first case, both parameters are dynamically scoped and have to be provided by the caller. In the second case, the parameter $?\mathsf{p}_1$ is available in the declaring scope and so it is (lexically) captured.

Although structural coeffects have more desirable syntactic properties, we aim to capture this non-principality too as it is practically useful – Haskell's implicit parameters use it and it can be used to model resource rebinding in distributed systems such as [54]).

## 3.6 BEYOND PASSIVE CONTEXTS

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

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

CALCULUS OF CAPABILITIES Crary et al. [14] introduced the Calculus of Capabilities to provide a sound system with region-based memory management for low-level code that can be easily compiled to assembly language. They build on the work of Tofte and Talpin [65] who developed an *effect system* (discussed in Section 2.2.2) that uses lexically scoped *memory regions* to provide an efficient and controlled memory management.

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

```
let calculate = λinput →
  letrgn ρ in
  let x = refρinput in !x
```

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

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

```
let calculate = λinput →
  letrgn ρ in
  let x = refρinput in x
```

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

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

SOFTWARE UPDATING Dynamic software updating (DSU) [21, 29] is the ability to update programs at runtime without stopping them. The Proteus system developed by Stoyle et al. [56] investigates what language support is needed to enable safe dynamic software updating in C-like languages. The system is based on the idea of capabilities.

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

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

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

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

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

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

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

## 3.7 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 [53, 36] (which also inspired the example presented in Chapter 1). His HOP language supports cross-compilation and programs execute in different contexts. However, HOP is not statically type checked.

In the software engineering community, a number of authors have addressed the problem of context-aware computations. Hirschfeld et al. propose *Context-Oriented Programming* (COP) as a methodology [30]. The COP paradigm has been later implemented by programming language features. Costanza [13] develops a domain-specific LISP-like language ContextL and Bardram [5] 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 [7, 17]. Here, the *context* focuses more on concrete physical context (obtained from the device sensors) than context as an abstract language feature.

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

## 3.8 summary

TODO

# BIBLIOGRAPHY

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[18] A. Filinski. 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.

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

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

[21] 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.

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

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

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

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

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

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

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

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

[35] 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.

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

[37] 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.

[38] 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.

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

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

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

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

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

[44] 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.

[45] D. Orchard. Programming contextual computations.

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

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

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

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

[50] 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.

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

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

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

[54] 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.

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

[56] 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.

[57] 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.

[58] 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.

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

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

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

[62] 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.

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

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

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

[66] 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.

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

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

[69] 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.

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

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

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

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

[74] 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.

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

[76] 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.

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

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