CONTENTS

1 WHY CONTEXT-AWARE PROGRAMMING MATTERS

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

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 [29], data-parallelism and also asynchronous computing [61].

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 [39] 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 [49].

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 [39], OpenCL or JavaScript [37]) 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 [25] which "uniquely identifies the framework API revision offered by a version of the Android platform". Most changes in the libraries (but not all) are additive.

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

For example, the Figure 1 shows an excerpt from the Http module in the F# Data library[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 [39] or F# quotations [59]) 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 [28].

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

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

1.1.4 *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, wrap-

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

1.2 TOWARDS CONTEXT-AWARE LANGUAGES

The four examples presented in the previous section cover different 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 [13], 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 [33] – the user has to explicitly annotate constants, but the rest of the information is inferred automatically.

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

fetchLocalNews

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

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

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 Γ, the expression e has a type τ. The variable context Γ is necessary to determine the type of expressions. Consider an expression $x + y$. In many languages, including Java, C# and F#, the type could be int 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:

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

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 practi-

cal 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 [24, 38], 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 [62], communication in message-passing systems [32] and atomicity in concurrent applications [21].

Type and effect systems are usually specified judgements of the form $\Gamma \vdash e : \alpha, \sigma$, meaning that the expression e has a type α in (free-variable) context Γ and additionally may have effects described by σ. Effect systems are typically added to a language that already supports effectful operations as a way of increasing the safety – the type and effect system provides stronger guarantees than a plain type system. Filinsky [19] refers to this approach as *descriptive*[1].

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

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

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

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

Figure 3: Simple effect system

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

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

Figure 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 [40] showed that effects (including partiality, exceptions, non-determinism and I/O) can be modelled uisng the category theoretic notion of *monad*.

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

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

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

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

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

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

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

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

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"* [40, 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"* [40, p. 5].

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

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

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

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

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

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

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

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

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

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

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

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

2.2.2 *Marriage of effects and monads*

The work on effect systems and monads both tackle the same problem – representing and tracking of computational effects. The two lines of research have been joined by Wadler and Thiemann [78]. 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 [63]) and further developed by Atkey [5] in his work on *parameterized monads*.

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

2.2.3 *Context-dependent languages and meta-languages*

The theoretical parts of this thesis extend the work of Uustalu and Vene who use comonads to give the semantics of data-flow computations [69] and more generally, notions of *context-dependent computations* [68]. The computations discussed in the latter work include streams, arrays and containers – this is a more diverse set of examples, but they all mostly represent forms of collections. Ahman et al. [3] discuss the relation between comonads and *containers* in more details.

The utility of comonads has been explored by a number of authors before. Brookes and Geva [11] use *computational* comonads for intensional semantics[3]. In functional programming, Kieburtz [34] proposed to use comonads for stream programming, but also handling of I/O and interoperability.

Biermann and de Paiva used comonads to model the necessity modality $\Box$ in intuitionistic modal S4 [10], linking programming languages derived from modal logics to comonads. One such language has been reconstructed by Pfenning and Davies [50]. Nanevski et al. extend this work to Contextual Modal Type Theory (CMTT) [43], which again shows the importance of comonads for *context-dependent* computations.

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

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

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

3 The structure of computational comonad has been also used by the author of this thesis to abstract evaluation order of monadic computations [48].

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

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

The definition is similar to monad with "reversed arrows". Intuitively, the counit operation extracts a value α from a value that carries additional context $C\alpha$. The cobind operation turns a context-dependent function $C\alpha \to \beta$ into a function that takes a value with context, applies the context-dependent function to value(s) in the context and then propagates the context. The next section makes this intuitive definition more concrete. More detailed discussion about comonads can be found in Orchard's PhD thesis [46].

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

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

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

In causal data-flow, we can interpret the list as a list consisting of past values, with the current value in the head. Then, the cobind operation 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 α, the expression **prev** e has also type α. The fact that the language models data-flow and values are lists (of past values) is a matter of semantics, which is defined as follows:

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

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

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

Figure 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. [41, 42] use the $\Box\alpha$ type to mean *mobile code*, that is code that can be evaluated at any node of a distributed system (the evaluation corresponds to the axiom $\Box\alpha \to \alpha$). Finally, Davies and Pfenning [16] consider staged computations and interpret $\Box\alpha$ as a type of (unevaluated) expressions of type α.

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

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

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

4 In fact, Pfenning and Davies [50, 43] never mention comonads explicitly. This is done in later work by Gabbay et al. [23], but the connection between the language and comonads is not as direct as in case of monadic or comonadic semantics covered in the last few pages.

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

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

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

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

The *meta-language* approach of Pfenning, Davies and Nanevski et al. is closely related to our work. Most importantly, Contextual Modal Type Theory (CMTT) uses indexed $\Box$ modality which seems to correspond to indexed comonads (in a similar way in which effect systems correspond to indexed monads). The relation between CMTT and comonads has been suggested by Gabbay et al. [23], but the meta-language employed by CMTT does not directly correspond to comonadic operations. For example, our let box typing rule from Figure 5 is not a primitive of CMTT and would correspond to $\mathsf{box}(\Psi, \mathsf{letbox}(e_1, x, e_2))$. Nevertheless, the indexing in CMTT provides a useful hint for adding indexing to the work of Uustalu and Vene.

2.3 THROUGH SUB-STRUCTURAL AND BUNCHED LOGICS

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

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

In this section, we look at type systems that reconsider Γ in a number of ways. First of all, sub-structural type systems [79] restrict the use of variables in the language. Most famously linear type systems introduced by Wadler [76] 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 [44], 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 [45] (starting a fruitful line of research in software verification), but it is also interesting on its own.

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

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

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

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

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

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

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

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 $\alpha \twoheadrightarrow \beta$ (corresponding to ";" and "," respectively). The key property is that only the first kind of functions can share variables with the context where a function is declared, while the second restricts such sharing. We do not attempt to give a detailed description here as it is not immediately to coeffects – for more information, refer to O'Hearn's introduction [44].

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

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

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

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

2.4 CONTEXT ORIENTED PROGRAMMING

The importance of context-aware computations is perhaps most obvious when considering mobile application, client/server web applications or even the internet of things. A pioneering work in the area using functional languages has been done by Serrano [54, 37] (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 [31]. The COP paradigm has been later implemented by programming language features. Costanza [14] develops a domain-specific LISP-like language ContextL and Bardram [6] proposes a Java framework for COP.

Finally, the subject of context-awareness has also been addressed in work focusing on the development of mobile applications [8, 18]. Here, the *con-*

text focuses more on concrete physical context (obtained from the device sensors) than context as an abstract language feature.

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

2.5 SUMMARY

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

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

3 CONTEXT-AWARE SYSTEMS

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

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

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

3.1 STRUCTURE OF COEFFECT SYSTEMS

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

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

3.1.1 *Lambda abstraction*

The difference between effects and coeffects becomes apparent when we consider lambda abstraction. The typical lambda abstraction rule for effect systems looks as (*abs-eff*) in Figure 8. Wadler and Thiemann [78] 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 [35], is a network of scientists sharing the same basic assumptions and techniques.

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

Figure 8: Lambda abstraction for pure and effectful computations

This is the key property of *output impurity*. The effects are only produced when the function is evaluated and so the effects of the body are attached to the function. A recent work by Tate [63] uses the term *producer* effect systems for such standard systems and characterises them as follows:

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

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

3.1.2 *Notions of context*

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

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

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

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

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

The interesting case is when we construct a lambda function $\lambda x.e$, marshall it and send it to another node. In that case, the context requirements can be satisfied in a number of ways. When the same resource is available at the target machine (e. g.. current time), we can send the function with a

context requirement and *rebind* the resource. However, if the resource is not available (e. g.. GPS on the server), we need to capture a *remote reference*.

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

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

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

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

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

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

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

3.1.3 *Scalars and vectors*

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

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

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

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

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

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

3.2 FLAT COEFFECT SYSTEMS

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

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

3.2.1 *Implicit parameters and type classes*

Haskell provides two examples of flat coeffects – type class and implicit parameter constraints [77, 36]. Both of the features introduce additional *constraints* on the context requiring that the environment provides certain operations for a type (type classes) or that it provides values for named implicit parameters. In the Haskell type system, constraints C are attached to the types of top-level declarations, such as let-bound functions. The Haskell notation $\Gamma \vdash e : C \Rightarrow \tau$ corresponds to our notation $\Gamma \circledast C \vdash e : \tau$.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 9: Coeffect rules for tracking implicit parameters

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

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

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

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

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

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

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

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

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

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

The semantics is defined inductively over the typing derivation:

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

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

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

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

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

Monadic semantics using the reader monads differs as follows:

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

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

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

Figure 10: Semantics of a language with implicit parameters

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

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

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

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

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

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

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

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

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

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

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

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

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

As we discuss later in Section 3.?, the reader monad can be extended to model rebinding. However, later examples in this chapter, such as liveness in Section 3.2.3 show that other context-aware computations cannot be captured by *any* monad.

TYPE CLASSES. Another type of constraints in Haskell that is closely related to implicit parameters are *type class* constraints [77]. They provide a principled form of ad-hoc polymorphism (overloading). When a code uses an overloaded operation (e.g. comparison or numeric operators) a constraint is placed on the context in which the operation is used. For example:

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

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

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

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

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

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

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

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

3.2.2 *Distributed computing*

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

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

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

A practical language that supports rebindable resources is for example Acute [55]. In the following example, we use the construct **access** Res to represent access to a rebindable resource named Res. The following simple function accessed a database and a current date and filters values based on the date:

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

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

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

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

Figure 11: Sample client-server application with input validation

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

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

CROSS-COMPILATION. A related issue with distributed programming is the need to target increasing number of diverse platforms. Modern applications often need to run on multiple platforms (iOS, Android, Windows or as JavaScript) or multiple versions of the same platform. Many programming languages are capable of targeting multiple different platforms. For example, functional languages that can be compiled to native code and JavaScript include, among others, F#, Haskell and OCaml [72].

Links [13], F# WebTools and WebSharper [60, 47], ML5 and QWeSST [41, 53] and Hop [37] go further and allow including code for multiple distinct platforms in a single source file. A single program is then automatically split and compiled to multiple target runtimes. This posses additional challenges – it is necessary to check where each part of the program can run and statically guarantee that it will be possible to compile code to the required target platform (safe *multi-targetting*).

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

Consider the client-server example in Figure 11. The retrieveProduct function represents the server-side, while showPrice is called on the client-side

a.) Set based type system for cross-compilation, inspired by Links [13]

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

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

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

b.) Version number based type system, inspired by Android API level [17]

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

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

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

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

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

In Links [13], functions can be annotated as client-side, server-side and database-side. F# WebTools [47] supports cross-compiled (mixed-side) functions similar to validateInput. However, these are single-purpose language features and they are not extensible. A practical implementation needs to be able to capture multiple different patterns – sets of environments (client, server, mobile) for distributed computing, but also Android API level [17] to cross-compile for multiple versions of the same platform.

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

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

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

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

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

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

In addition to the work discussed already, ML5 [41] is another important work that looks at tracking of execution environments. It uses modalities of modal S4 to represent the environment – this approach is similar to coeffects, both from the practical perspective, but also through deeper theoretical links. We return to this topic in Chapter **??**.

3.2.3 *Liveness analysis*

Live variable analysis (LVA) [4] is a standard technique in compiler theory. It detects whether a free variable of an expression may be used by a program during its evaluation (it is *live*) or whether it is definitely not needed (it is *dead*). As an optimization, compiler can remove bindings to dead variables as they are never accessed. Wadler [75] describes the property of a variable that is dead as the *absence* of a variable.

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 13: Liveness annotations with sequential and pointwise composition

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

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

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

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

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

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

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

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

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

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

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

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

Figure 14: Coeffect rules for tracking whole-context liveness

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

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

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

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

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

$$[\![\Gamma \mathbin{@} \mathsf{D} \vdash c_i : \tau_i]\!] = \lambda() \to \delta(c_i) \qquad (\textit{const})$$

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

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

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

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

$$[\![\Gamma \mathbin{@} r \vdash e_1\ e_2 : \tau_2]\!] = \lambda x \to \mathbf{let}\ g = [\![\Gamma \mathbin{@} r \vdash e_1 : \tau_1 \xrightarrow{\mathsf{D}} \tau_2]\!]\ x\ \mathbf{in}\ g\ () \qquad (\textit{app-1})$$

$$[\![\Gamma \mathbin{@} \mathsf{L} \vdash e_1\ e_2 : \tau_2]\!] = \lambda x \to \mathbf{let}\ g = [\![\Gamma \mathbin{@} \mathsf{L} \vdash e_1 : \tau_1 \xrightarrow{\mathsf{L}} \tau_2]\!]\ x\ \mathbf{in}\ g\ ([\![\Gamma \mathbin{@} \mathsf{D} \vdash e_2 : \tau_1]\!]\ ()) \qquad (\textit{app-2})$$

$$[\![\Gamma \mathbin{@} r \sqcup (s \sqcap t) \vdash e_1\ e_2 : \tau_2]\!] = \lambda x \to \mathbf{let}\ g = [\![\Gamma \mathbin{@} r \vdash e_1 : \tau_1 \xrightarrow{t} \tau_2]\!]\ x\ \mathbf{in}\ g\ ([\![\Gamma \mathbin{@} s \vdash e_2 : \tau_1]\!]\ x) \qquad (\textit{app-3})$$

Figure 15: Semantics of a language with liveness analysis

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

$$(\lambda x \to 42)\ y \qquad (1)$$
$$\mathsf{twoTimes}\ 42 \qquad (2)$$
$$(\lambda x \to x)\ 42 \qquad (3)$$

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

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

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

However, ordinary option type is not sufficient. We need to capture the fact that the representation depends on the annotation – in other words,

the type is *indexed* by the coeffect annotation. The indexing is discussed in details in Section X. For now, it suffices to define the semantics using two separate rules:

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

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

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

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

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

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

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

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

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

3.2.4 *Data-flow languages*

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

Lucid [74] is a declarative data-flow language designed by Wadge and Ashcroft. In Lucid, variables represent streams and programs are written as transformations over streams. A function application square(x) represents a stream of squares calculated from the stream of values x.

The data-flow approach has been successfully used in domains such as development of real-time embedded application where many *synchronous languages* [7] build on the data-flow paradigm. The following example is inspired by the Lustre [26] language and implements program to count the number of edges on a Boolean stream:

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

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

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

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

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

The semantics of Lucid-like languages can be captured using a number of mathematical structures. Wadge [73] originally proposed to use monads, while Uustalu and Vene later used comonads [67]. In Chapter **??**, we extend the latter approach. However, the present chapter presents a sketch of a data-flow semantics defined directly on streams.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$[\![\Gamma \circledast n+1 \vdash \textbf{prev}\ e : \tau]\!] = \lambda\langle \mathbf{v}_0, \ldots, \mathbf{v}_{n+1}\rangle \to [\![\Gamma \circledast n \vdash e : \tau]\!]\ \langle \mathbf{v}_1, \ldots, \mathbf{v}_{n+1}\rangle \qquad (\textit{prev})$$

$$[\![\Gamma \circledast n \vdash e : \tau]\!] = \lambda\langle \mathbf{v}_0, \ldots, \mathbf{v}_n\rangle \to [\![\Gamma \circledast n' \vdash e : \tau]\!]\ \langle \mathbf{v}_0, \ldots, \mathbf{v}_{n'}\rangle \qquad (\textit{sub})$$

$$[\![\Gamma \circledast n \vdash \lambda y.e : \tau_1 \xrightarrow{n} \tau_2]\!] = \lambda\langle \mathbf{v}_0, \ldots \mathbf{v}_n\rangle \to \lambda(y, g) \to [\![\Gamma, y : \tau_1 \circledast n \vdash e : \tau_2]\!]\ \langle(\mathbf{v}_0, y_0), \ldots, (\mathbf{v}_n, y_n)\rangle \qquad (\textit{abs})$$

$$\begin{aligned}[\![\Gamma \circledast \max(m, n+p) \vdash e_1\ e_2 : \tau_2]\!] = \lambda(\mathbf{v}_0, \ldots, \mathbf{v}_{\max(m,n+p)}) \to \\ \textbf{let}\ g = [\![\Gamma \circledast m \vdash e_1 : \tau_1 \xrightarrow{p} \tau_2]\!]\ (\mathbf{v}_0, \ldots, \mathbf{v}_m) \\ \textbf{in}\ g\ (\ [\![\Gamma \circledast n \vdash e_2 : \tau_1]\!]\ (\mathbf{v}_0, \ldots, \mathbf{v}_n), \ldots, \\ \qquad [\![\Gamma \circledast n \vdash e_2 : \tau_1]\!]\ (\mathbf{v}_p, \ldots, \mathbf{v}_{n+p})\ )\end{aligned} \qquad (\textit{app})$$

Figure 17: Semantics of a simple data-flow language

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

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

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

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

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

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

Uustalu and Vene [68] model causal data-flow computations using a non-empty list NeList $\tau = \tau \times (\text{NeList}\ \tau + 1)$ over the input. A function $\tau_1 \to \tau_2$ is thus modelled as NeList $\tau_1 \to \tau_2$. This model is difficult to implement efficiently, as it creates unbounded lists of past elements.

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

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

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

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

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

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

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

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

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

3.2.5 *Permissions and safe locking*

In the implicit parameters and data-flow examples, the context provides additional resources or values that may be accessed at runtime. However, it may also track *permissions* or *capabilities* to perform some operation. Liveness can be seen as a trivial example – when the context is live, it contains a permission to access variables. In this section, we briefly consider a system for safe locking of Flanagan and Abadi [20] as one, more advanced example. Calculus of capabilities of Cray et al. [15] is discussed later in Section 3.4.

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

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

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

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

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

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

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

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

3.3 STRUCTURAL COEFFECT SYSTEMS

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

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

3.3.1 *Liveness analysis revisited*

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

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

$$\textbf{let}\ \mathsf{constant} = \lambda y \to \lambda x \to y$$
$$\textbf{let}\ \mathsf{answer} = (\lambda x \to x)\ 42$$

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

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

- The fact that variables are now a vector means that we cannot freely reorder them. This guarantees that $x:\tau, y:\tau @ \langle \mathsf{L}, \mathsf{D}\rangle$ can not be confused with $y:\tau, x:\tau @ \langle \mathsf{L}, \mathsf{D}\rangle$. We need to define the type system in a way that is similar to sub-structural systems (discussed in Section 2.3) which provide explicit rules for manipulating the context.
- We choose to attach a vector of annotations to a vector of variables, rather than attaching individual annotations to individual variables. This lets us unify and combine flat and structural systems as discussed in Chapter **??**, but the alternative is briefly explored in Chapter **??**.

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

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

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

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

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

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

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

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

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

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

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

b.) Structural rules for context manipulation

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

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

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

Figure 18: Structural coeffect liveness analysis

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

a.) Semantics of ordinary expressions

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

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

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

$$[\![\Gamma_1, \Gamma_2 \mathbin{@} \mathbf{r} \times (\mathsf{L} \sqcup \mathbf{s}) \vdash e_1\ e_2 : \tau_2]\!] = \lambda(\mathbf{v_1}, \mathbf{v_2}) \to \textbf{let}\ g = [\![\Gamma_1 \mathbin{@} \mathbf{r} \vdash e_1 : \tau_1 \xrightarrow{\mathsf{L}} \tau_2]\!]\ \mathbf{v_1}\ \textbf{in}\ g\ ([\![\Gamma_2 \mathbin{@} \mathbf{s} \vdash e_2 : \tau_1]\!]\ \mathbf{v_2}) \qquad (\textit{app-1})$$

$$[\![\Gamma_1, \Gamma_2 \mathbin{@} \mathbf{r} \times (\mathsf{D} \sqcup \mathbf{s}) \vdash e_1\ e_2 : \tau_2]\!] = \lambda(\mathbf{v_1}, \mathbf{v_2}) \to \textbf{let}\ g = [\![\Gamma_1 \mathbin{@} \mathbf{r} \vdash e_1 : \tau_1 \xrightarrow{\mathsf{D}} \tau_2]\!]\ \mathbf{v_1}\ \textbf{in}\ g\ () \qquad (\textit{app-2})$$

b.) Semantics of structural context manipulation

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

$$[\![\Gamma_1, y{:}\tau, x{:}\tau', \Gamma_2 \mathbin{@} \mathbf{r} \times \langle t, s \rangle \times \mathbf{q} \vdash e : \sigma]\!] = \lambda(\mathbf{v_1}, y, x, \mathbf{v_2}) \to [\![\Gamma_1, x{:}\tau', y{:}\tau, \Gamma_2 \mathbin{@} \mathbf{r} \times \langle s, t \rangle \times \mathbf{q} \vdash e : \sigma]\!]\ (\mathbf{v_1}, x, y, \mathbf{v_2}) \qquad (\textit{exch})$$

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

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

Figure 19: Semantics of structural liveness

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

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

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

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

Ordinary rules

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

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

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

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

$$(let)\ \frac{\Gamma_1, v : \sigma\,@\,\mathbf{r} \times \langle t\rangle \vdash e_1 : \tau \quad \Gamma_2\,@\,\mathbf{s} \vdash e_2 : \sigma}{\Gamma_1, \Gamma_2\,@\,\mathbf{r} \times (t * \mathbf{s}) \vdash \textbf{let}\ x = e_2\ \textbf{in}\ e_1 : \tau}$$

Structural rules

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

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

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

Figure 20: Coeffect rules for tracking implicit parameters

SUMMARY. The structural coeffect system for tracking liveness analysis semantics is "the other way round"

3.3.2 *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 $!_{k_1} A_1, ..., !_{k_n} A_n$ is written as $\tau_1, ..., \tau_n\,@\,\langle k_1, ..., k_n\rangle$. This difference is further explained in Section **??**.

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 Γ (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 **??** 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 Γ 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 Γ_2 S-times means that, in total, the variables in Γ_2 will be used $(t * S)$-times. Recall that $t * S$ is a scalar multiplication of a vector. Meanwhile, the variables in Γ_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:

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

To avoid name clashes, we α-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:

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

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 β-reduction.

3.3.3 *Data-flow languages (revisited)*

When discussing data-flow languages in the previous section, we said that the context provides past values of variables. 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.

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 21, 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 Γ_2. Thus, we need $t + S$ past values of Γ_2 to perform the call (e. g., we need $1 + S$ values to get 1 past value of the input σ, $2 + S$ values to get 2 past values of σ, *etc.*).

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

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

The expression always requires the previous value of y and adds it to a previous value of the parameter x. Evaluating the value of the argument **prev** (**prev** y) requires two past values of y and so the overall requirement is 3 past values:

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

The derivation uses (*app*) to get requirements $\langle 1,3\rangle$ and then (*contr*) to take the maximum, showing three past values are sufficient. Reducing the expression by substitution we get **prev** $(y + (\textbf{prev}\ (\textbf{prev}\ y)))$. Semantically, this

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

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

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

Figure 21: type and coeffect system for dataflow caching

performs stream lookups $y[1] + y[3]$ where the indices are the number of enclosing **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.

3.3.4 *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 [12], 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 [70] and a well known attack known as SQL injection [28, 27]. In the latter chase, we want to check that SQL commands cannot be directly constructed from, potentially dangerous, inputs provided by the user. Consider the type checking of the following expression in a context containing variables `id` and `msg`:

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

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

In runtime checking, all (string) values need to be wrapped in an object that stores Boolean flag (for tainting) or more complex data (for provenance). In static checking, the information need to be associated with the variables in the variable context. We use tainting as a motivating example for *structural* coeffects in Section X.

3.3.5 *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. [71] provide the first (provably) sound type system that guarantees non-inference and Sabelfeld et al. [52] survey more recent work. The checking of information flows has been also integrated (as a single-purpose extension) in the FlowCaml [56] language. Finally, Russo et al. and Swamy et al. [51, 58] 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* [64], which detects which parts of programs can be partially evaluated (do not depend on user input) and *program slicing* [65] that identifies parts of programs that contribute to the output of an expression.

3.4 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. [15] 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 [66] 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 = ref_ρ input 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 = ref_ρ input 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) [22, 30] is the ability to update programs at runtime without stopping them. The Proteus system developed by Stoyle et al. [57] 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.

3.5 SUMMARY

TODO

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

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

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

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

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

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

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

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

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

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

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

[12] 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.

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

[14] 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.

[15] 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.

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

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

[18] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[36] 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.

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

[38] 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.

[39] 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.

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

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

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

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

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

[45] 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.

[46] D. Orchard. Programming contextual computations.

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

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

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

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

[51] 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.

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

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

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

[55] 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.

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

[57] 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.

[58] 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.

[59] 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.

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

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

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

[63] 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.

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

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

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

[67] 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.

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

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

[70] 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.

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

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

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

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

[75] 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.

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

[77] 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.

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

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