CONTENTS

1 INTRODUCTION

Some intro

1.1 HOW LAMBDA WORKS

The key difference - how lambda works

Effect systems, introduced by Gifford and Lucassen [22], track *effects* of computations, such as memory access or message-based communication [27]. Their approach augments typing judgments with effect information: $\Gamma \vdash e : \tau, F$. Wadler and Thiemann explain how this shapes effect analysis of lambda abstraction [67]:

> *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.*

In contrast to the static analysis of *effects*, the analysis of *context-dependence* does not match this pattern. In the systems we consider, lambda abstraction places requirements on both the *call-site* (latent requirements) and the *declaration-site* (immediate requirements), resulting in different syntactic properties. We informally discuss three examples first that demonstrate how contextual requirements propagate. Section ? then unifies these in a single calculus.

Also [52]

> *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.*

Demonstrate using distributed computations
Demonstrate using dataflow/liveness
Perhaps mention LINQ as a motivation for cross-compilation.. [32]

1.2 ASSOCIATING WITH CONTEXT

-> Type providers make this important

1.3 FLAT COEFFECT SYSTEM

resources?

1.4 STRUCTURAL COEFFECT SYSTEM

The *flat coeffect system* presented in the previous sections has a number of uses, but often we need to track context-dependence in a more fine-grained

way. To track neededness or security, we need to associate information with individual *variables* of the context.

At the same time, we want to avoid developing multiple variants of coeffect systems – indeed, our motivation is to develop a *single unified mechanism* for tracking context-dependence. In this section, we present a more powerful *structural coeffect calculus*, which is a generalization of the flat calculus.

1.4.1 *Motivation: Tracking array accesses*

Similarly to the flat version, the *structural coeffect calculus* works with contexts and functions annotated with a coeffectt tags, written $C^r\Gamma$ and $C^r\tau_1 \to \tau_2$, respectively, but we use richer tag structure.

As an example, consider a language that allows us to get a value of a variable (representing some changing data-source) x versions back using the syntax $a_{[x]}$. To track information about individual variables, we use a product-like operation $\times$ on tags to mirrors the product structure of variables. For example:

$$C^{5\times 10}(a : \mathsf{D}_{\mathsf{nat}}, b : \mathsf{D}_{\mathsf{nat}}) \vdash a_{[5]} + b_{[10]} : \mathsf{nat}$$

The coeffect tag 5×10 corresponds to the free-variable context a, b, denoting that we need at most 5 and 10 past values of a and b. If we substitute c for both a and b, we need another operation to combine multiple tags associated with a single variable:

$$C^{5\ \mathsf{max}\ 10}(c : \mathsf{D}_{\mathsf{nat}}) \vdash c_{[5]} + c_{[10]} : \mathsf{nat}$$

In this example, the operation max would be the *max* function and so $5\ \mathsf{max}\ 10 = 10$. Before looking at the formal definition, consider the typing of let bindings:

$$\textbf{let}\ c = \textbf{if}\ \mathsf{test}()\ \textbf{then}\ a\ \textbf{else}\ b$$
$$a_{[15]} + c_{[10]}$$

The expression has free variables a and b (we ignore test, which is not a data source). It defines c, which may be assigned either a or b. The variable a may be used directly (second line) or indirectly via c.

The expression assigned to c uses variables a and b, so its typing context is $C^{0\times 0}(a, b)$. The value 0 is the unit of max and it denotes empty coeffect. The typing context of the body is $C^{15\times 10}(a, c)$.

To combine the tags, we take the coeffect associated with c and apply it to the tags of the context in which c was defined using the max operation. This is then combined with the remaining tags from the body yielding the overall context: $C^{15\times(10\ \mathsf{max}\ (0\times 0))}(a, (a, b))$. Using a simple normalization mechanism (described later), this can be further reduced to $C^{(15\ \mathsf{max}\ 10)\times 10}(a, b)$. This gives us the required information – we need at most $max(15, 10)$ past values of a and at most 10 past values of b.

1.5 CONTRIBUTIONS

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 2.1), then we look at coeffects as the dual of effect systems (Section 2.2) and extend the duality to category theory, looking at the categorical dual of monads known as *comonads* (Section 2.3). Finally we look at logically inspired type systems that are closely related to our structural coeffects (Section 2.4).

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

2.1 THROUGH APPLICATIONS

The general theme of this thesis is improving programming languages to better support writing *context-dependent* (or *context-aware*) computations. With current trends in the computing industry such as mobile and ubiquitous computing, this is becoming an important topic. In software engineering and programming community, a number of authors have addressed this problem from different perspectives. Hirschfeld et al. propose *Context-Oriented Programming* (COP) as a methodology [26], and the subject has also been addressed in mobile computations [7, 17]. In programming languages, Costanza [13] develops a domain-specific LISP-like language ContextL and Bardram [5] proposes a Java framework for COP.

We approach the problem from a different perspective, building on the tradition of statically-typed functional programming languages and their theories. However, even in this field, there is a number of calculi or language features that can be viewed as context-dependent.

2.1.1 *Motivation for flat coeffects*

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

IMPLICIT PARAMETERS In Haskell, implicit parameters [29] are a special kind of variables that may behave as dynamically scoped. This means, if a function uses parameter ?p, then the caller of the function must define ?p and set its value. Implicit parameters can be used to parameterise a computation (involving a chain of function calls) without passing parameters explicitly as additional arguments of all involved functions. A simple language with implicit parameters has an expression ?p to read a parameter value and an expression[1] **letdyn** ?p = e_1 **in** e_2 that sets a parameter ?p to the value of e_1 and evaluates e_2 in a context containing ?p

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

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

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

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

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

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

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

This way of assigning type to format and calling it is not the only possible, though. We could also say that the outer function requires both of the implicit parameters and the result is a (pure) function with no context requirements. This interaction between implicit parameters and lambda abstraction demonstrates one of the key aspects of coeffects and will be discussed later. Implicit parameters will also sever as one of our examples in Chapter Y.

TYPE CLASSES Implicit parameters are closely related to *type classes* [66]. In Haskell, type classes provide a principled form of ad-hoc polymorphism (overloading). When a code uses an overloaded operation (e.g. comparison or numeric operators) a constraint is placed on the context in which the operation is used. For example:

```
twoTimes :: Num α ⇒ α → α
twoTimes x = x + x
```

The constraint Num a on the function type arises from the use of the + operator. From the implementation perspective, the type class constraint means

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

that the function takes a hidden parameter – a dictionary that provides the operation $+ :: \alpha \to \alpha \to \alpha$. Thus, the type $\mathsf{Num}\ \alpha \Rightarrow \alpha \to \alpha$ can be viewed as $(\mathsf{Num}_\alpha \times \alpha) \to \alpha$. Implicit parameters work in exactly the same way – they are passed around as hidden parameters.

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

REBINDABLE RESOURCES The need for parameters that do not strictly follow static scoping rules also arises in distributed computing. This problem has been addressed, for example, by Bierman et al. and Sewell et al. [8, 46]. To quote the first work: *"Dynamic binding is required in various guises, for example when a marshalled value is received from the network, containing identifiers that must be rebound to local resources."*

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

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

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

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

DISTRIBUTED COMPUTING AND MULTI-TARGETTING An increasing number of programming languages is capable of running across multiple different platforms or execution environments. Functional programming languages that can be compiled to JavaScript (to target web and mobile clients) include, among others, F#, Haskell and OCaml [61].

Links [12], F# libraries [50, 40], ML5 and QWeSST [34, 45] and Hop [30] go further and allow a single source program to be compiled to multiple target runtimes. This posses additional challenges – it is necessary to track where each part of computation runs and statically guarantee that it will be possible to compile code to the required target platform (safe *multi-targetting*).

We demonstrate the problem by looking at input validation. In distributed applications that communicate over unsecured HTTP channel, user input needs to be validated interactively on the client-side (to provide immedi-

ate response) and then again on the server-side (to guarantee safety). For example:

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

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

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

In Links [12], functions can be annotated as client-side, server-side and database-side. F# WebTools [40] adds functions that support multiple targets (mixed-side). However, these are single-purpose language features and they are not extensible. For example, in modern mobile development it is also important to track minimal supported version of runtime[2].

Requirements on the execution environment can be viewed as contextual properties, but could be also presented as effects (use of some API required only in certain environment is a computational effect). We discuss the difference in Section X. Furthermore, the theoretical foundations of distributed languages like ML5 [34] suggest that a contextual treatment is more appropriate. We return to ML5 when discussing semantics in Section 2.3.3.

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

Data-flow languages provide a different example. Lucid [63] 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 $square(a)$ represents a stream of squares calculated from the stream of values a.

The data-flow approach has been successfully used in domains such as development of real-time embedded application where many *synchronous languages* [6] build on the data-flow paradigm. The following example is inspired by the Lustre [23] language and implements program to count the number of edges on a Boolean stream:

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

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

The construct **prev** x returns a stream consisting of previous values of the stream x. The second value of **prev** x is first value of x (and the first value is undefined). The construct y **fby** x returns a stream whose first element is the first element of y and the remaining elements are values of x. Note that in

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

Lucid, the constants such as false and 0 are constant streams. Formally, the construct are defined as follows (writing x_n for n-th element of a stream x):

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

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

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

To understand the nature of the context, we later look at the semantics of Lucid. This can be captured using a number of mathematical structures. Wadge [62] originally proposed to use monads, while Uustalu and Vene later used comonads [56].

2.1.2 *Motivation for structural coeffects*

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

LIVENESS ANALYSIS *Live variable analysis* (LVA) [3] is a standard technique in compiler theory. It detects whether a free variable of an expression may be used by a program later (it is *live*) or whether it is definitely not needed (it is *dead*). As an optimization, compiler can remove bindings to dead variables as the result is never accessed. Wadler [64] describes the property of a variable that is dead as the *absence* of a variable.

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

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

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

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

Static analyses can be classified as either *forward* or *backward* (depending on how they propagate information) and as either *must* or *may* (depending on what properties they guarantee). Liveness is a *backward* analysis – this means that the requirements propagates from variables to their declaration

sites. The distinction between *must* and *may* is apparent when we look at an example with conditionals:

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

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

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

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

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

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

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

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

Statically typed systems that based on tainting have been use to prevent cross-site scripting attacks [59] and a well known attack known as SQL injection [25, 24]. In the latter chase, we want to check that SQL commands cannot be directly constructed from, potentially dangerous, inputs provided by the user. Consider the type checking of the following expression in a context containing variables id and msg:

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

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

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

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

SECURITY AND CORE DEPENDENCY CALCULUS The checking of tainting is a special case of checking of the *non-interference* property in *secure information flow*. Here, the aim is to guarantee that sensitive information (such as credit card number) cannot be leaked to contexts with low secrecy (e.g. sent via an unsecured network channel). Volpano et al. [60] provide the first (provably) sound type system that guarantees non-inference and Sabelfeld et al. [44] survey more recent work. The checking of information flows has been also integrated (as a single-purpose extension) in the FlowCaml [47] language. Finally, Russo et al. and Swamy et al. [43, 49] 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* [53], which detects which parts of programs can be partially evaluated (do not depend on user input) and *program slicing* [54] that identifies parts of programs that contribute to the output of an expression.

2.1.3 *Beyond passive contexts*

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

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

CALCULUS OF CAPABILITIES Crary et al. [14] introduced the Calculus of Capabilities to provide a sound system with region-based memory management for low-level code that can be easily compiled to assembly language.

They build on the work of Tofte and Talpin [55] who developed an *effect system* (discussed in Section 2.3.2) that uses lexically scoped *memory regions* to provide an efficient and controlled memory management.

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

$$\begin{array}{l}\textbf{let}\ \mathsf{calculate} = \lambda \mathsf{input} \rightarrow \\ \quad \textbf{letrgn}\ \rho\ \textbf{in} \\ \quad \textbf{let}\ \mathsf{x} = \textbf{ref}_{\rho}\,\mathsf{input}\ \textbf{in}\ !\mathsf{x}\end{array}$$

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:

$$\begin{array}{l}\textbf{let}\ \mathsf{calculate} = \lambda \mathsf{input} \rightarrow \\ \quad \textbf{letrgn}\ \rho\ \textbf{in} \\ \quad \textbf{let}\ \mathsf{x} = \textbf{ref}_{\rho}\,\mathsf{input}\ \textbf{in}\ \mathsf{x}\end{array}$$

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) is the ability to update programs at runtime without stopping them.

Stoyle et al. [48].

THESIS PERSPECTIVE ?

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

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

Figure 1: Simple effect system

2.2 THROUGH TYPE AND EFFECT SYSTEMS

Introduced by Gifford and Lucassen [22, 31], 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 [51], communication in message-passing systems [27] and atomicity in concurrent applications [20].

Type and effect systems are usually specified judgements of the form $\Gamma \vdash e : \alpha, \sigma$, meaning that the expression e has a type 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 [18] refers to this approach as *descriptive*[3].

SIMPLE EFFECT SYSTEM The structure of a simple effect system is demonstrated in Figure 1. 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 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 2 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.

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

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

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

Figure 2: Simple effect system

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 1, 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).

2.3 THROUGH LANGUAGE SEMANTICS

Another pathway to coeffects leads through the semantics of effectful and context-dependent computations. In a pioneering work, Moggi [33] 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.

2.3.1 *Effectful languages and meta-languages*

Moggi uses monads to define two formal systems. In the first formal system, a monad is used to model the *language* itself. This means that the semantics of a language is given in terms of a one specific monad and the semantics

can be used to reason about programs in that language. To quote *"When reasoning about programs one has only one monad, because the programming language is fixed, and the main aim is to prove properties of programs"* [33, 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"* [33, 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[4].

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

4 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.

pret 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.

2.3.2 *Marriage of effects and monads*

The work on effect systems and monads both tackle the same problem – representing and tracking of computational effects. The two lines of research have been joined by Wadler and Thiemann [67]. 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 [52]) and further developed by Atkey [4] in his work on *parameterized monads*.

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

2.3.3 *Context-dependent languages and meta-languages*

The theoretical parts of this thesis extend the work of Uustalu and Vene who use comonads to give the semantics of data-flow computations [58] and more generally, notions of *context-dependent computations* [57]. The computations discussed in the latter work include streams, arrays and containers – this is a more diverse set of examples, but they all mostly represent forms

of collections. Ahman et al. [2] discuss the relation between comonads and *containers* in more details.

The utility of comonads has been explored by a number of authors before. Brookes and Geva [10] use *computational* comonads for intensional semantics[5]. In functional programming, Kieburtz [28] proposed to use comonads for stream programming, but also handling of I/O and interoperability.

Biermann and de Paiva used comonads to model the necessity modality $\Box$ in intuitionistic modal S4 [9], linking programming languages derived from modal logics to comonads. One such language has been reconstructed by Pfenning and Davies [42]. Nanevski et al. extend this work to Contextual Modal Type Theory (CMTT) [36], which again shows the importance of comonads for *context-dependent* computations.

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

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

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

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

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

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

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

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 [39].

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.

5 The structure of computational comonad has been also used by the author of this thesis to abstract evaluation order of monadic computations [41].

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

Figure 3: Typing for a comonadic language with contextual staged computations

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

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

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

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

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

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).

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[6].

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. [34, 35] 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 [15] 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 3 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.
- (*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.

6 In fact, Pfenning and Davies [42, 36] never mention comonads explicitly. This is done in later work by Gabbay et al. [21], 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.

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

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

Figure 4: Exchange, weakening and contraction typing rules

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. [21], but the meta-language employed by CMTT does not directly correspond to comonadic operations. For example, our let box typing rule from Figure 3 is not a primitive of CMTT and would correspond to $\mathsf{box}(\Psi, \mathsf{letbox}(e_1, x, e_2))$. Nevertheless, the indexing in CMTT provides a useful hint for adding indexing to the work of Uustalu and Vene.

2.4 THROUGH 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 [68] restrict the use of variables in the language. Most famously linear type systems introduced by Wadler [65] 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 [37], the variable context is a tree formed by multiple different constructors (e.g. one that allows sharing and one that does not). Most importantly, bunched typing has contributed to the development of separation logic [38] (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 4. 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

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

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

Figure 5: Exchange, weakening and contraction rules for bunched typing

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:

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

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

One important note about bunched typing is that it requires a different interpretation. The omission of weakening and contraction in linear logic means that variable can be used exactly once. In bunched typing, variables may still be duplicated, but only using the ";" separator. The type system can be interpreted as specifying whether a variable may be shared between the body of a function and the context where a function is declared. The system introduces two distinct function types $\alpha \to \beta$ 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 [37].

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.

2.5 SUMMARY

Oops!

BIBLIOGRAPHY

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[23] 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.

[24] 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.

[25] 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.

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

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

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

[29] 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.

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

[31] 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.

[32] 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.

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

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

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

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

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

[38] 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.

[39] D. Orchard. Programming contextual computations.

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

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

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

[43] 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.

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

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

[46] 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.

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

[48] 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.

[49] 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.

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

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

[52] 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.

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

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

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

[56] 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.

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

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

[59] 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.

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

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

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

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

[64] 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.

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

[66] 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.

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

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