CONTEXT-AWARE PROGRAMMING LANGUAGES

TOMAS PETRICEK

Computer Laboratory
University of Cambridge

2014

ABSTRACT

The development of programming languages needs to reflect important changes in the industry. In recent years, this included e. g. the development of parallel programming models (in reaction to the multi-core revolution). This thesis is a response to another such revolution – the diversification of devices and systems where programs run.

We develop the foundations for statically typed functional languages that understand the *context* or environment in which programs execute. Such context includes different platforms (and their versions) in cross-platform applications, resources available in different execution environments (e. g. GPS sensor on a phone and database on the server), but also more traditional notions such as variable context (tracking variable usage in static analyses) or past values in stream-based data-flow programming.

The thesis presents three *coeffect* calculi that capture different notions of context-awareness: *flat* calculus capturing contextual properties of the execution environment, *structural* calculus capturing contextual properties related to variable usage and *meta-language* calculus which allows reasoning about multiple notions of context-dependence in a single language and provides pathway to embedding the other two calculi in existing languages.

Although the focus of this thesis is on the syntactical properties of the presented systems, we also discuss their category-theoretical motivation. We introduce the notion of an *indexed* comonad (the dual of monads) and show how they provide semantics of the presented three calculi.

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.

1.1 PROGRAMMING LANGUAGE AND INNOVATION

We briefly consider two practical concerns that led to the development of new programming languages. This helps to explain why context-aware programming is of importance, which we cover in the next section. The examples are by no means representative, but they illustrate the 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 [7], data-parallelism and also asynchronous computing [17].

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 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 [11] 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.

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[1].

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 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 [14].

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 3 and Chapter **??**, we develop two simple calculi to understand and capture the structure of the problems and, finally, Chapter **??** unifies the two abstractions.

1.2 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 [11], OpenCL or JavaScript [10]) a part of the compilation (e. g. SQL generation) 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.

1.2.1 *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 [5] 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[2]. 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

2 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[3]: *"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.

1.2.2 *Context awareness #2: System capabilities*

Another example related to the previous one is when libraries use metaprogramming techniques (such as LINQ [11] or F# quotations [16]) 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.

1.2.3 *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.

3 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 [6].

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.

1.2.4 *Context-awareness #4: Access patterns in data-flow languages*

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 $x_{[a]}$. To track information about individual variables, we use a product-like operation $\times$ on tags to mirrors the product structure of variables. For example:

?

The annotations 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`, respectively. If we substitute `c` for both `a` and `b`, we need to combine the context-requirements and take the maximum of the requirements of the individual variables:

?

This version of the calculus removes the non-determinism of the lambda abstraction from the previous version. As we associate information with individual variables, lambda abstraction creates a function that requires the context requirements associated with the variable that is being abstracted over.

For example, if we wrapped the earlier example above in a function taking `a` (and using `b` from the declaration-site) then the function would have context requirements 5 – that is the number associated with the variable `a`.

1.2.5 *Context-awareness #4: Resource & data availability*

A vast majority of applications accesses some data sources (like database) or resources (like GPS sensor on a phone). This is more tricky for client/server applications where a part of program runs on the server-side and another part runs on the client-side. I believe that these two parts should be written as a single program that is cross-compiled to two parts (and I tried to make that possible with F# Web Tools [**?**]) a long time ago; more recently [WebSharper](http://websharper.com/) implemented a similar idea).

So, say we have a function `validateInput`, `readData` and `displayMessage` in my prorgam. I want to look at their types and see what resources (or *context*) they require. For example, `readData` requires *database* (or perhaps a database with a specific name), `displayMessage` requires access to *user interface* and `validateInput` has no special requirements.

This means that I can call `validateInput` from both server-side and client-side code - it is safe to share this piece of code, because it has no special requirements. However, when I write a code that calls all three functions without any remote calls, it will only run on a thick client that has access to a database as well as user interface.

I'll demonstrate this idea with a sample (pseudo-)code in a later section, so do not worry if it sounds a bit abstract at first.

1.3 COEFFECTS: TOWARDS CONTEXT-AWARE LANGUAGES

The above examples cover a couple of different scenarios, but they share a common theme - they all talk about some *context* in which an expression is evaluated. The context has essentially two aspects:

- **Flat context** represents additional data, resources and meta-data that are available in the execution environment (regardless of where in the program you access them). Examples include resources like GPS sensors or databases, battery status, framework version and similar.
- **Structural context** contains additional meta-data related to variables. This can include provenance (source of the variable value), usage information (how often is the value accessed) or security information (does it contain sensitive data).

As a proponent of statically typed functional languages I believe that a context-aware programming language should capture such context information in the type system and make sure that basic errors (like the ones demonstrated in the four examples above) are ruled out at compile time.

This is essentially the idea behind *coeffects*. Let's look at an example showing the idea in (a very simplified) practice and then I'll say a few words about the theory (which is the main topic of this thesis).

1.3.1 *Case study: Coeffects in action*

So, how should a context-aware language look? Surely, there is a wide range of options, but I hope I convinced you that it needs to be *context-aware* in some way! I'll write my pseudo-example in a language that looks like F#, is fully statically typed and uses type inference.

I think that type inference is particularly important here - we want to check quite a few properties that should not that difficult to infer (If I call

a function that needs GPS, I need to have GPS access!) Writing all such information by hand would be very cumbersome.

So, let's say that we want to write a client/server news reader where the news are stored in a database on a server. When a client (iPhone or Windows phone) runs, we get GPS location from the phone and query the server that needs to connect to the "News" database using a password defined somewhere earlier (perhaps loaded from a server-side config file):

```
let lookupNews(location) =
  let db = query("News", password)
  selectNews(db, location)

let readNews() =
  let loc = gpsLocation()
  remote {
    lookupNews(loc)
  }

let iPhoneMain() =
  createCocoaWidget(readNews)

let windowsMain() =
  createMetroWidget(readNews)
```

The idea is that `lookupNews` is a server-side function that queries the "News" database based on the specified `location`. This is called from the client-side by `readNews` which get the current GPS position and uses a `remote { .. }` block to invoke the `lookupNews` function remotely (how exactly would this be written is a separate question - but imagine a simple REST request here).

Then, we have two main functions, `iPhoneMain` and `windowsMain` that will serve as two entry points for iPhone and Windows build of the client-side application. They are both using a corresponding platform-specific function to build the user interface, which takes a function for reading news as an argument.

If you wanted to write and compile something like this today, you could use F# in Xamarin Studio to target iPhone and Window phone, but you'd either need two separate end-application projects or a large number of unmaintainable `#if` constructs. Why not just use a single project, if the application is fairly simple?

I imagine that a context-aware statically typed language would let you write the above code and if you inspected the types of the functions, you would see something like this:

```
password      :   string { sensitive }
lookupNews    :   Location -{ database }-> list<News>

gpsLocation   :   unit -{ gps }-> Location
readNews      :   unit -{ rpc, gps }-> Async<list<News>>

iPhoneMain    :   unit -{ cocoa, gps, rpc }-> unit
windowsMain   :   unit -{ metro, gps, rpc }-> unit
```

The syntax is something that I just made up for the purpose of this article - it could look different. Some information could even be mapped to other

visual representations (e.g. blueish background for the function body in your editor). The key thing is that we can learn quite a lot about the context usage:

- `password` is available in the context, but is sensitive and so we cannot return it as a result from a function that is called via an RPC call. - `lookupNews` requires database access and so it can only run on the server-side or on a thick client with local copy of the database. - `gpsLocation` accesses GPS and since we call it in `readNews`, this function also requires GPS (the requirement is propagated automatically). - We can compile the program for two client-side platforms - the entry points require GPS, the ability to make RPC calls and Cocoa or Metro UI platform, respectively.

When writing the application, I want to be always able to see this information (perhaps similarly to how you can see type information in the various F# editors). I want to be able to reference multiple versions of base libraries - one for iPhone and another for Windows and see all the API functions at the same time, with appropriate annotations. When a function is available on both platforms, I want to be able to reuse the code that calls it. When some function is available on only one platform, I want to solve this by designing my own abstraction, rather than resorting to ugly `#if` pragmas.

Then, I want to take this single program (again, strutured using whatever abstractions I find appropriate) and compile it. As a result, I want to get a component (containing `lookupNews`) that I can deploy to the server-side and two packages for iPhone and Windows respectively, that reference only one or the other platform.

1.3.2 *Coeffects: Theory of context dependence*

If you're expecting a "Download!" button or (even better) a "Buy now!" button at the end of this article, then I'll disappoint you. I have no implementation that would let you do all of this. My work in this area has been (so far) on the theoretical side. This is a great way to understand what is *actually* going on and what does the *context* mean. And if you made it this far, then it probably worked, because I understood the problem well enough to be write a readable article about it!

1.3.2.1 *Brief introduction to type systems*

I won't try to reproduce the entire content of the thesis in this introduction – but I will try to give you some background in case you are interested (that should make it easier to look at the papers above). We'll start from the basics, so readers familiar with theory of programming languages can skip to the next section.

Type systems are usually described in the form of *typing judgement* that have the following form:

$$\Gamma \vdash e : \tau \tag{1}$$

The judgement means that, given some variables described by Γ, the expression or program e has a type τ. What does this mean? For example, what is a type of the expression `x + y`? Well, this depends - in F# it could be some numeric type or even a string, depending on the types of `x` and

y. That's why we need the variable context Γ which specifies the types of variables. So, for example, we can have:

$$x : int, y : int \vdash x + y : int \tag{2}$$

Here, we assume that the types of `x` and `y` (on the left hand side) are both `int` and as a result, we derive that the type of `x + y` is also an `int`. This is a valid typing, for the expression, but not the only one possible - if `x` and `y` were of type `string`, then the result would also be `string`.

1.3.2.2 *Checking what program does with effect systems*

Type systems can be extended in various interesting ways. Essentially, they give us an approximation of the possible values that we can get as a result. For example refinement types [**?**] can estimate numerical values more precisely (e.g. less than 100). However, it is also possible to track what a program does - how it *affects* the world. For example, let's look at the following expression that prints a number:

$$x : int \vdash print\ x : unit \tag{3}$$

This is a reasonable typing in F# (and ML languages), but it ingores the important fact that the expression has a *side-effect* and prints the number to the console. In Haskell, this would not be a valid typing, because `print` would return an `IO` computation rather than just plain `unit` (for more information see IO in Haskell [4].

However, monads are not the only way to be more precise about side-effects. Another option is to use effect system [**?**] which essentially annotates the result of the typing judgement with more information about the *effects* that occur as part of evaluation of the expression:

$$x : int \vdash print\ x : unit\&\{IO\} \tag{4}$$

The effect annotation is now part of the type - so, the expression has a type `unit & { io }` meaning that it does not return anything useful, but it performs some I/O operation. Note that we do not track what *exactly* it does - just some useful over-approximation. How do we infer the information? The compiler needs to know about certain language primitives (or basic library functions). Here, `print` is a function that performs I/O operation.

The main role of the type system is dealing with composition - so, if we have a function `read` that reads from the console (I/O operation) and a function `send` that sends data over network, the type system will tell us that the type and effects of `send (read ())` are `unit & io, network`.

Effect systems are a fairly established idea - and they are a nice way to add better purity checking to ML-like languages like F#. However, they are not that widely adopted (interestingly, checked exceptions in Java are probably the most major use of effect system). However, effect systems are also a good example of general approach that we can use for tracking contextual information...

4 http://www.haskell.org/haskellwiki/IO_inside

1.3.2.3 *Checking what program requires with coeffect systems*

How could we use the same idea of *annotating* the types to capture information about the context? Let's look at a part of the program from the case study that I described earlier:

$$pass : string \vdash query(\text{"News"}, pass) : NewsDb \tag{5}$$

The expression queries a database and gets back a value of the `NewsDb` type (for now, let's say that `"News"` is a constant string and `query` behaves like the SQL type provider in F# [5] and generates the `NewsDb` type automatically).

What information do we want to capture? First of all, we want to add an annotation saying that the expression requires *database access*. Secondly, we want to mark the `pass` variable as *secure* to guarantee that it will not be sent over an unsecured network connection etc. The *coeffect typing judgement* representing this information looks like this:

$$pass : string^{\{secure\}}@\{database\} \vdash query(\text{"News"}, pass) : NewsDb \tag{6}$$

Rather than attaching the annotations to the *resulting type*, they are attached to the variable *context*. In other words, the equation is saying – given a variable `pass` that is marked as secure and additional environment providing database access, the expression `query("News", pass)` is well typed and returns a `NewsDb` value.

As a side-note, it is well known that *effects* correspond to *monads* (and Haskell uses monads as a way of implementing limited effect checking). Quite interestingly, *coeffects* correspond to the dual concept called *comonads* and, with a syntactic extension akin to the `do` notation or *computation expressions* [**?**], you could capture contextual propreties by adding comonads to a language.

1.4 SUMMARY

I started by explaining the motivation for my work - different problems that arise when we are writing programs that are aware of the context in which they run. The context includes things such as execution environment (databases, resources, available devices), platform and framework (different versions, different platforms) and meta-data about data we access (sensitivity, security, provenance).

This may not be preceived as a major problem - we are all used to write code that deals with such things. However, I believe that the area of *context-aware* programming is a source of many problems and pains - and programming languages can help!

In the second half of the article, I gave a brief introduction to *coeffects* – a programming language theory that can simplify dealing with context. The key idea is that we can use types to track and check additional information about the *context*. By propagating such information throughout the program (using type system that is aware of the annotations), we can make sure that none of the errors that I used as a motivation for this article happen.

5 http://www.pinksquirrellabs.com/post/2013/12/09/The-Erasing-SQL-type-provider.aspx

2 INTRODUCTION

In the rest of the chapter, we first look at the distinguishing factor of two of the presented coeffect systems (how they differ from other existing work). Then we look at a number of concrete problems that are discussed in greater details in later chapters. We use the examples to demonstrate the real-world problems that motivate our work. (Theoretical and other motivations are discussed later in Chapter **??**).

2.1 CONTEXT AND LAMBDA ABSTRACTION

Our work on context-aware programming languages connects two directions in existing research on the theory of programming languages. On one side, effect systems [4] and monadic computations [12, 23] provide a detailed and established method for tracking what effects programs have – that is, how they affect the environment where they execute. On the other side, the work on comonadic notions of computations [20] shows how to use the mathematical dual of monads – comonads – to give categorical semantics of context-dependent computations.

Effect systems introduced track actions such as memory operations or communication. They are described by typing judgments of a form $\Gamma \vdash e : \alpha, \sigma$ where Γ is the context of a program (typically available variables), e is the expression (program) itself, α is the type of values returned by the program (e.g. integer or boolean) and σ is a set of possible effects. The judgment states that, given the context Γ, an expression has a type α and can only perform effects specified by the set σ. Wadler and Thiemann [23] explain how this shapes effect analysis of a lambda abstraction – that is, how effect systems analyze the effects associated with a definition of a function:

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

This means that, when a programmer defines a function, the system records that executing the function will perform the effects of the body of the function. However, simply defining a function is an effect-free computation. Tate [19] calls such effect systems *producer* effect systems and generalizes the idea of function to more general "thunking":

> *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.*

In contrast to the static analysis of (producer) effect systems, 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.

2.2 WHY COEFFECTS MATTER

This section gives three examples of context-dependent computations whose properties can be captured by the tree calculi presented in this thesis. We look at an example of the *flat coeffect calculus* (Chapter 3), *structural coeffect calculus* (Chapter **??**) and *coeffect meta-language* (Chapter **??**).

2.2.1 *Flat coeffect calculus*

The flat coeffect calculus associates a single piece of information with the context. As an example, we look at a simple distributed programming language that includes the concept of *resources*. A resource may be accessed using a special construct **access** Res. The following example shows a function taht lists recent events – it accesses the resource News (representing a database) and a resource Clock (with the current time):

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

Consider a scenario where the function is *declared* on the server-side and then transferred and *executed* on the client-side. The resource News represents a database that is only available on the server-side and so the function needs to keep a remote reference to the server. However, the Clock resource may (or may not) be available on the client-side. If the resource is available on the client, then it may be re-bound and the function will use the current client's data – for example, to accommodate for time-zone changes.

This example demonstrates how lambda abstraction behaves for context-dependent computations. The context requirement of the function body is a set of resources $\{\mathsf{Clock}, \mathsf{News}\}$. The context requirements are split between the *declaration-site* and *call-site*. However, there are multiple possible splittings. The splitting $\{\mathsf{News}\} \cup \{\mathsf{Clock}\}$ models the case when the database is accessed from the server, but time is taken from the client, while the splitting $\{\mathsf{Clock}, \mathsf{News}\} \cup \{\}$ corresponds to the case when both resources are accessed from the server.

2.2.2 *Structural coeffect calculus*

The calculus used in the previous section annotates the entire context with a single value – such as the set of required resources. However, sometimes it is desirable to annotate not the entire context, but individual variables of the context.

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 $\mathsf{a}_{[\mathsf{x}]}$. To track information about individual variables, we use a product-like operation $\times$ on tags to mirrors the product structure of variables. For example:

$$(\mathsf{a} : \mathsf{stream}, \mathsf{b} : \mathsf{stream}) \mathbin{@} 5 \times 10 \vdash \mathsf{a}_{[5]} + \mathsf{b}_{[10]} : \mathsf{nat}$$

The annotations 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, respectively. If we

substitute c for both a and b, we need to combine the context-requirements and take the maximum of the requirements of the individual variables:

$$(\mathsf{c} : \mathsf{stream})\ @\ \mathsf{max}(5, 10) \vdash \mathsf{c}_{[5]} + \mathsf{c}_{[10]} : \mathsf{nat}$$

This version of the calculus removes the non-determinism of the lambda abstraction from the previous version. As we associate information with individual variables, lambda abstraction creates a function that requires the context requirements associated with the variable that is being abstracted over.

For example, if we wrapped the earlier example above in a function taking a (and using b from the declaration-site) then the function would have context requirements 5 – that is the number associated with the variable a.

2.2.3 *Coeffect meta-language*

TODO: Give some example that uses the coeffects metalanguage style. This will probably be meta-programming with open expressions (similarly to what Pfenning and Nanevski do with contextual modal type theory).

2.3 WHY CONTEXT MATTERS

We claimed earlier that context-dependent comptuations are becoming increasingly common and our work is focused on annotating the (free-variable) context with additional information. The importance of context can be also demonstrated by looking at a technology that focuses solely on making the context richer – F# type providers.

TODO: Say more about type providers – they extend the context Γ so that it can be lazily loaded and it can be huge. Potentially, it could be also annotated with additional meta-data...

2.4 CONTRIBUTIONS

TODO: We present three calculi that model common notions of context-dependence and can be used as basis for developing context-aware programming langauges with static type systems.

3 STRUCTURAL COEFFECT LANGUAGE

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.

3.1 INTRODUCTION

3.1.1 *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:

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

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:

$$C^{5\vee 10}(c : \mathsf{stream}) \vdash c_{[5]} + c_{[10]} : \mathsf{nat}$$

In this example, the operation $\vee$ would be the *max* function and so $5 \vee 10 = 10$. 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 $\vee$ 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 $\vee$ operation. This is then combined with the remaining tags from the body yielding the overall context: $C^{15\times(10\vee(0\times 0))}(a, (a, b))$. Using a simple normalization mechanism (described later), this can be further reduced to $C^{(15\vee 10)\times 10}(a, b)$. 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.

3.1.2 *Structural coeffect tags*

In the previous section, the $\vee$ operation behaves similarly to the flat $\vee$ operation. However, the type system does not require some of the semilattice properties, because some uses are replaced with the $\times$ operation. We do not require any properties about the $\times$ operation. For example, in the previous example cannot be commutative (since a tag 15×10 has different meaning than 10×15). However, we relate the operations using distributivity laws to allow normalization that was hinted above.

Definition 1. *A* structural coeffect tag structure $(S, \times, \vee, 0, 1)$ *is a tuple where* $(S, \vee)$ *is a lattice-like structure with unit* 0*. The additional structure is formed by a binary operation* $\times$ *and element* $1 \in S$ *such that for all* $r, s, t \in S$*, the following equalities hold:*

$$r \vee (s \vee t) = (r \vee s) \vee t \qquad \text{(associativity)}$$
$$r \vee s = s \vee r \qquad \text{(commutativity)}$$
$$r \vee 0 = r \qquad \text{(lower bound)}$$
$$r \vee (s \times t) = (r \vee s) \times (r \vee t) \qquad \text{(distributivity)}$$
$$1 \vee r = 1 \qquad \text{(upper bound)}$$

The tag 0 represents that no coeffect is associated with a variable (i.e when a variable is always accessed using standard variable access). The tag 1 is used to annotate empty variable context. For example, the context of an expression $\lambda x.x$ is empty and it needs to carry an annotation. We explain exactly how this works when we introduce the type system. The fact that 1 is the upper bound means that combining it with other coeffect annotations does not affect it and so empty contexts cannot carry any information.

The structure generalizes the *flat coeffect tag structure* introduced in Section **??**, but it additionally requires a special element 1 representing the upper bound. Given a flat coeffect structure, we can construct a structural coeffect structure (but not the other way round). For certain structures, the element 1 may be already present, but in general, it can be added as a new element. This construction will be important in Section **??**, where we show that λ_{Cs} calculus generalizes λ_{Cf}.

Lemma 1. *A flat coeffect tag structure* $(S, \vee, 0)$ *implies a structural coeffect tag structure.*

Proof. Take $1 \notin S$, then $(S \cup \{1\}, \vee, \vee, 0, 1)$ is a structural coeffect tag structure. From the properties of flat coeffect tag structure, we get that the $\vee$ operation is associative, commutative and 0 is the unit with respect to $\vee$.

To prove the distributivity, we need to show $r \vee (s \vee t) = (r \vee s) \vee (r \vee t)$. This easily follows from commutativity, associativity and idempotence of the flat coeffect tag structure. □

3.1.3 *Structural coeffect type system*

The simply typed system for λ_{Cs} uses the same syntax of types as the system for λ_{Cf}. The rules are of a form $C^r\Gamma \vdash e : \tau$ where r is a tag provided by a *structural coeffect tag structure* $(S, \times, \vee, 0)$.

The system differs from λ_{Cf} in a significant aspect. It contains explicit structural rules that manipulate with the context. Such rules allow reordering, duplicating and other manipulations with variable context. Such rules are known from affine or linear type systems where they are removed to

obtain more restrictive system. In our system, the rules are present, but they manipulate the variable structure Γ as well as the associated tag structure r.

As in linear and affine systems, the variable context Γ in our system is not a simple set. Instead, we use the following tree-like structure (which is more similar to bunched types than to linear or affine systems):

$$\Gamma ::= () \mid (x : \tau) \mid (\Gamma, \Gamma)$$

The syntax $()$ represents an empty context, so the structure defines a binary trees where leaves are either variables or empty. Contexts such as $C^{1\times(r\times 1)}((), (x, ()))$ contain unnecessary number of empty contexts $()$. However, we need to construct them temporarily, because certain rules require splitting a context and, by our definition, the context $(x : \tau)$ is not splitable.

The typing rules of the system are shown in Figure **??**. Many of the structural rules are expressed in terms of a helper judgement $C^r\Gamma \Rightarrow_c C^r\Gamma$.

STANDARD RULES. Variable access (λ_{Cs} -Tvar) annotates the corresponding variable with an empty coeffect 0. The λ_{Cs} -Tfun rule assumes that the context of the body can be split into the variable of the function and other (potentially empty) context and it attaches the coeffect associated with the function variable to the resulting function type $C^s\tau_1 \to \tau_2$.

The λ_{Cs} -Tapp rule combines coeffects s, t, r associated with the function-returning expression, argument and the function type respectively. The result of evaluating the argument in the context t is passed to the function that requires context r, so the variables used in the context Γ_2 are annotated with the combination of coeffects $r \vee t$. The variable context required to evaluate the function value is independent and so it is annotated just with coeffects s. Finally, the λ_{Cs} -Tlet rule is derived from let binding and application, but we show it separately to aid the understanding.

STRUCTURAL RULES. The remaining rules are not syntax-directed. They are embedded using the λ_{Cs} -Tctx rule and expressed using a helper judgement $C^{r_1}\Gamma_1 \Rightarrow_c C^{r_2}\Gamma_2$ that says that the context on the left-hand side can be transformed to the context on the right-hand side. The transformation can be applied to any part of the context, which is captured using the λ_{Cs} -Tnest rule (it is sufficient to apply the transformation on the left part of the context; the right part can be transformed using λ_{Cs} -Texch).

The λ_{Cs} -Tempty rule allows attaching empty context to any existing context. The rule is needed to type-check lambda abstractions that do not capture any outer variable. The λ_{Cs} -Tweak rule is similar, but it represents *weakening* where an unused variable is added. The associated coeffect tag does not associate context information with the variable. This is needed to type-check lambda abstraction that does not use the argument.

The λ_{Cs} -Tcontr rule is a limited form of contraction. When a variable appears repeatedly, it can be reduced to a single occurrence. The rule is needed to satisfy the side condition of λ_{Cs} -Tfun when the body uses the argument repeatedly. The two associativity rules together with λ_{Cs} -Texch provide ways to rearrange variables. Finally, λ_{Cs} -Tsub represents sub-coeffecting – in λ_{Cs} the rule operates on coeffects of individual variables.

3.1.4 *Properties of reductions*

Similarly to the flat version, the λ_{Cs} calculus is defined abstractly. We cannot define its operational meaning, because that will differ for every con-

TYPING RULES $C^r\Gamma \vdash e : \alpha$

$$(var)\ \frac{}{C^e(v : \alpha) \vdash v : \alpha}$$

$$(const)\ \frac{c : \alpha \in \Phi}{C^1() \vdash c : \alpha}$$

$$(fun)\ \frac{C^{r\times s}(\Gamma, v : \alpha) \vdash e : \beta \qquad v \notin \Gamma}{C^r\Gamma \vdash \lambda v.e : C^s\alpha \to \beta}$$

$$(app)\ \frac{C^s\Gamma_1 \vdash e_1 : C^r\alpha \to \beta \qquad C^t\Gamma_2 \vdash e_2 : \alpha}{C^{s\times(r\oplus t)}(\Gamma_1, \Gamma_2) \vdash e_1\ e_2 : \beta}$$

$$(let)\ \frac{C^{r\times s}(\Gamma_1, v : \alpha) \vdash e_1 : \beta \qquad C^t\Gamma_2 \vdash e_2 : \alpha}{C^{r\times(s\oplus t)}(\Gamma_1, \Gamma_2) \vdash \mathbf{let}\ x = e_2\ \mathbf{in}\ e_1 : \beta}$$

$$(ctx)\ \frac{C^r\Gamma \vdash e : \alpha \qquad C^{r'}\Gamma' \Rightarrow_c C^r\Gamma}{C^{r'}\Gamma' \vdash e : \alpha}$$

CONTEXT RULES $C^r\Gamma \Rightarrow_c C^r\Gamma$

$$(nest)\ \frac{C^{r'}\Gamma_1' \Rightarrow_c C^r\Gamma_1}{C^{r'\times s}(\Gamma_1', \Gamma_2) \Rightarrow_c C^{r\times s}(\Gamma_1, \Gamma_2)}$$

$$(nest)\ C^{r\times s}(\Gamma_1, \Gamma_2) \Rightarrow_c C^{s\times r}(\Gamma_2, \Gamma_1)$$

$$(empty)\ C^{r\times 1}(\Gamma, ()) \Rightarrow_c C^r\Gamma$$

$$(weak)\ C^{r\times 0}(\Gamma, x : \alpha) \Rightarrow_c C^r\Gamma$$

$$(contr)\ C^{r+s}(x : \alpha) \Rightarrow_c C^{r\times s}(x : \alpha, x : \alpha)$$

$$(assoc)\ C^{r\times(s\times t)}(\Gamma_1, (\Gamma_2, \Gamma_3)) \Rightarrow_c C^{(r\times s)\times t}((\Gamma_1, \Gamma_2), \Gamma_3)$$

$$(sub)\ C^r\Gamma \Rightarrow_c C^s\Gamma \quad (\text{when } s \leqslant r)$$

Figure 2: Type system for the structural coeffect language λ_{sc}

crete application. For example, when tracking array accesses, variables are interpreted as arrays and $a_{[n]}$ denotes access to a specified element.

Just like previously, we can state general properties of the reductions. As the syntax of expressions is the same for λ_{Cs} as for λ_{Cf}, the substitution and reduction $\twoheadrightarrow_\beta$ are also the same and can be found in Figure **??**.

The structural coeffect calculus λ_{Cs} associates information with individual variables. This means that when an expression requires certain context, we know from what scope it comes – the context must be provided by a scope that defines the associated variable, which is either a lambda abstraction or global scope. This distinguishes the structural system from the flat system where context could have been provided by any scope and the lambda rule allowed arbitrary splitting of context requirements between the two scopes (or declaration and caller site).

INTERNALIZED SUBSTITUTION. Before looking at properties of the evaluation, we consider let binding, which can be viewed as internalized substitution. The typing rule λ_{Cs} -Tlet can be derived from application and abstraction as follows.

Lemma 2 (Definition of let binding)**.** *If* $C^r\Gamma \vdash (\lambda x.e_2)\ e_1 : \tau_2$ *then* $C^r\Gamma \vdash$ **let** $x = e_1$ **in** $e_2 : \tau_2$.

Proof. The premises and conclusions of a typing derivation of $(\lambda x.e_2)\ e_1$ correspond with the typing rule λ_{Cs} -Tlet:

$$\frac{\dfrac{C^{r\times s}(\Gamma_1, v : \tau_1) \vdash e_2 : \tau_2 \qquad v \notin \Gamma_1}{C^r\Gamma_1 \vdash \lambda v.e_2 : C^s\tau_1 \to \tau_2} \qquad C^t\Gamma_2 \vdash e_1 : \tau_1}{C^{r\times(s\vee t)}(\Gamma_1, \Gamma_2) \vdash (\lambda v.e_2)\ e_1 : \tau_2}$$ □

The term e_2 which is substituted in e_1 is checked in a different variable and coeffect context $C^t\Gamma_2$. This is common in sub-structural systems where a variable cannot be freely used repeatedly. The context Γ_2 is used in place of the variable that we are substituting for. The let binding captures substitution for a specific variable (the context is of a form $C^{r\times s}\Gamma, v : \tau$). For a general substitution, we need to define the notion of context with a hole.

SUBSTITUTION AND HOLES. In λ_{Cs}, the structure of the variable context is not a set, but a tree. When substituting for a variable, we need to replace the variable in the context with the context of the substituted expression. In general, this can occur anywhere in the tree. To formulate the statement, we define contexts with holes, written $\Delta[-]$. Note that there is a hole in the free variable context and in a corresponding part of the coeffect tag:

$$\begin{array}{rcll} \Delta[-] & ::= & C^1() & \\ & | & C^r(x : \tau) & \\ & | & C^-(-) & \\ & | & C^{r_1\times r_2}(\Gamma_1, \Gamma_2) & \text{(where } C^{r_i}\Gamma_i \in \Delta[-]) \end{array}$$

Assuming we have a context with hole $C^r\Gamma \in \Delta[-]$, the hole filling operation $C^r\Gamma[r'|\Gamma']$ fills the hole in the variable context with Γ' and the corresponding coeffect tag hole with r'. The operation is defined in Figure **??**. Using contexts with holes, we can now formulate the general substitution lemma for λ_{Cs}.

Lemma 3 (Substitution Lemma)**.** *If* $C^r\Gamma[R|v : \tau'] \vdash e : \tau$ *and* $C^S\Gamma' \vdash e' : \tau'$ *then* $C^r\Gamma[R \vee S|\Gamma'] \vdash e[v \leftarrow e'] : \tau$.

$$
\begin{aligned}
C^1()[r'|\Gamma'] &= C^1() \\
C^r(x:\tau)[r'|\Gamma'] &= C^r(x:\tau) \\
C^-(-)[r'|\Gamma'] &= C^{r'}\Gamma' \\
C^{r_1\times r_2}(\Gamma_1,\Gamma_2)[r'|\Gamma'] &= C^{r_1'\times r_2'}(\Gamma_1',\Gamma_2') \\
&\quad \text{where } C^{r_i'}\Gamma_i' = C^{r_i}\Gamma_i[r'|\Gamma']
\end{aligned}
$$

Figure 3: The definition of hole filling operation for $\Delta[-]$

Proof. Proceeds by rule induction over $\vdash$ using the properties of structural coeffect tag structure $(S, \vee, 0, \times, 1)$ (see Appendix **??**). □

Theorem 1 (Subject reduction)**.** *If* $C^r\Gamma \vdash e_1 : \tau$ *and* $e_1 \twoheadrightarrow_\beta e_2$ *then* $C^r\Gamma \vdash e_2 : \tau$.

Proof. Direct consequence of Lemma **??** (see Appendix **??**). □

LOCAL SOUNDNESS AND COMPLETENESS. As with the previous calculus, we want to guarantee that the introduction and elimination rules (λ_{Cs} -Tfun and λ_{Cs} -Tapp) are appropriately strong. This can be done by showing *local soundness* and *local completeness*, which correspond to β-reduction and η-expansion. Former is a special case of subject reduction and the latter is proved by a simple derivation:

Theorem 2 (Local soundness)**.** *If* $C^r\Gamma \vdash (\lambda x.e_2)\ e_1 : \tau$ *then* $C^r\Gamma \vdash e_2[x \leftarrow e_1] : \tau$.

Proof. Special case of subject reduction (Theorem **??**). □

Theorem 3 (Local completeness)**.** *If* $C^r\Gamma \vdash f : C^s\tau_1 \to \tau_2$ *then* $C^r\Gamma \vdash \lambda x.fx : C^s\tau_1 \to \tau_2$.

Proof. The property is proved by the following typing derivation:

$$
\dfrac{\dfrac{C^r\Gamma \vdash f : C^s\tau_1 \to \tau_2 \qquad C^0(x:\tau_1) \vdash x : \tau_1}{C^{r\times(s\vee 0)}(\Gamma, x:\tau_1) \vdash f\ x : \tau_2}}{C^r\Gamma \vdash \lambda x.fx : C^s\tau_1 \to \tau_2}
$$

□

In the last step, we use the *lower bound* property of structural coeffect tag, which guarantees that $s \vee 0 = s$. Recall that in λ_{Cf}, the typing derivation for $\lambda x.fx$ required for local completeness was not the only possible derivation. In the last step, it was possible to split the coeffect tag arbitrarily between the context and the function type.

In the λ_{Cs} calculus, this is not, in general, the case. The $\times$ operator is not required to be associative and to have units and so a unique splitting may exist. For example, if we define $\times$ as the operator of a *free magma*, then it is invertible and for a given t, there are unique r and s such that $t = r \times s$. However, if the $\times$ operation has additional properties, then there may be other possible derivation.

3.2 SEMANTICS OF STRUCTURAL COEFFECTS

The semantics of structural coeffect calculus λ_{Cs} can be defined similarly to the semantics of λ_{Cf}. The most notable difference is that the structure of coeffect tag now mirrors the structure of the variable context. Thus an expression $C^{r\times s}(\Gamma_1, \Gamma_2) \vdash e : \tau$ is modelled as a function $C^{r\times s}(\Gamma_1 \hat{\times} \Gamma_2) \to \tau$.

$$C^{r_1 \times \ldots \times r_n}(x_1 : \tau_1 \times \ldots \times x_n : \tau_n) \vdash e : \tau : C^{r_1 \times \ldots \times r_n}(\tau_1 \times \ldots \times \tau_n) \to \tau$$

$$\begin{array}{rcll}
C^0\Gamma \vdash x_i : \tau_i & = & \epsilon_0 & \lambda_C \\
C^r\Gamma \vdash \lambda x.e : C^s\tau_1 \to \tau_2 & = & \Lambda(C^{r\times s}(\Gamma, x : \tau_1) \vdash e : \tau_2 \circ_{r,s}) & \lambda_C \\
C^{s\times(r\vee t)}(\Gamma_1, \Gamma_2) \vdash e_1\ e_2 : \tau & = & (\lambda(\gamma_1, \gamma_2) \to C^s\Gamma_1 \vdash e_1 : C^r\tau_1 \to \tau_2\ \gamma_1\ (C^t\Gamma_2 \vdash e_2 : \tau_1 t, r\ \gamma_2)) \circ_{s,(r\vee t)} & \lambda_C \\
C^{r'}\Gamma' \vdash e : \tau & = & C^r\Gamma \vdash e : \tau \circ C^{r'}\Gamma' \Rightarrow_c C^r\Gamma & \lambda_C
\end{array}$$

$$\begin{array}{rcll}
C^{r'\times s}(\Gamma_1', \Gamma_2) \Rightarrow_c C^{r\times s}(\Gamma_1, \Gamma_2) & = & {}_{r,s} \circ (C^{r'}\Gamma_1' \Rightarrow_c C^r\Gamma_1 \times \mathsf{id}) \circ_{r',s} & \lambda_{Cs}\ \text{-SCr} \\
C^{r\times s}(\Gamma_1, \Gamma_2) \Rightarrow_c C^{s\times r}(\Gamma_2, \Gamma_1) & = & {}_{s,r} \circ \mathsf{swap} \circ_{r,s} & \lambda_{Cs}\ \text{-SCe} \\
C^{r\times 1}(\Gamma, ()) \Rightarrow_c C^r\Gamma & = & \mathsf{fst} \circ_{r,1} & \lambda_{Cs}\ \text{-SCe} \\
C^{r\times 0}(\Gamma, x : \tau) \Rightarrow_c C^r\Gamma & = & \mathsf{fst} \circ_{r,0} & \lambda_{Cs}\ \text{-SCw} \\
C^{r\vee s}(x : \tau) \Rightarrow_c C^{r\times s}(x : \tau, x : \tau) & = & {}_{r,s} \circ \Delta_{r,s} & \lambda_{Cs}\ \text{-SCc} \\
C^{r\times(s\times t)}(\Gamma_1, (\Gamma_2, \Gamma_3)) \Rightarrow_c C^{(r\times s)\times t}((\Gamma_1, \Gamma_2), \Gamma_3) & = & {}_{r\times s,t} \circ ({}_{r,s} \times \mathsf{id}) \circ \mathsf{assoc}_1 \circ (\mathsf{id} \times_{s,t}) \circ_{r,s\times t} & \lambda_{Cs}\ \text{-SCa} \\
C^r\Gamma \Rightarrow_c C^s\Gamma & = & \iota_{r,s} & \lambda_{Cs}\ \text{-SCs}
\end{array}$$

Figure 4: Categorical semantics for λ_{Cs}

As discussed in **??**, the variable context Γ in structural coeffect system is not a simple finite product, but instead a binary tree. To model this, we do not use ordinary products in the domain of the semantic function, but instead use a special constructor $\hat{\times}$. This way, we can guarantee that the variable structure corresponds to the tag structure.

3.2.1 *Structural tagged comonads*

To model composition of functions, we reuse the definition of *tagged comonads* from Section **??** without any change. This means that composing morphisms $T^r\tau_1 \to \tau_2$ with $T^s\tau_2 \to \tau_3$ still gives us a morphism $T^{r\vee s}\tau_1 \to \tau_3$ and we use the $\vee$ operation to combine the context-requirements.

However, functions that do not exist in context have only a single input variable (with a single corresponding tag). To model complex variable contexts, we need two additional operations that allow manipulation with the variable context. Similarly to the model of λ_{Cf}, we also require operations that model duplication and sub-coeffecting:

Definition 2 (Structural tagged comonad)**.** *Given a structural coeffect tag structure* $(S, \times, \vee, 0, 1)$ *a* structural tagged comonad *is a tagged comonad over* $(S, \vee, 0)$ *comprising of* T^r, ϵ_0 *and* $(-)r, s$ *together with a mapping* $-\hat{\times}-$ *from a pair of objects* $\times$ *to an object and families of mappings:*

$$\begin{array}{rcl}
{}_{r,s} & : & T^rA \times T^sB \to T^{(r\times s)}(A \hat{\times} B) \\
{}_{r,s} & : & T^{(r\times s)}(A \hat{\times} B) \to T^rA \times T^sN
\end{array}$$

And with a family of mappings $\iota_{r,s} : T^rA \to T^sA$ *for all* $r, s \in S$ *such that* $r \vee s = r$.

The family of mappings $\iota_{r,s}$ is the same as for *flat* coeffects and it can still be used to define a family of mappings that represents *duplicating* of variables while splitting the additional coeffect tags:

$$\Delta_{r,s} : T^{(r \vee s)} A \to T^r A \times T^s A$$
$$\Delta_{r,s}(\gamma) = (\iota_{(r \vee s),r}\ \gamma, \iota_{(r \vee s),s}\ \gamma)$$

The type of the $_{r,s}$ operation looks similar to the one used for *flat* coeffects, but with two differences. Firstly, it combines tags using $\times$ instead of $\vee$, which corresponds to the fact that the variable context now consists of two parts (a tree node). Secondly, to model the tree node, the resulting context is modelled as $A \hat{\times} B$ (instead of $A \times B$ as previously).

To model structural coeffects, we also need $_{r,s}$, which serves as the dual of $_{r,s}$. It represents *splitting* of context containing multiple variables. The operation was not needed for λ_{Cf}, because there *splitting* could be defined in terms of *duplication* provided by $\Delta_{r,s}$. For λ_{Cs}, the situation is different. The $_{r,s}$ operation takes a context annotated with $r \times s$ that carries $A \hat{\times} B$.

Examples of *structural tagged comonads* are shown in Section **??**. Before looking at them, we finish our discussion of categorical semantics.

CATEGORICAL NOTES. The mapping T^r can be extended to an endofunctor $\hat{T}^r$ in the same way as in Section **??**. However, we still cannot freely manipulate the variables in the context. Given a context modelled as $T^{r \times s}(A \hat{\times} B)$, we can lift a morphism f to $\hat{T}^{r \times s}(f)$, but we cannot manipulate the variables, because $A \hat{\times} B$ is not a product and does not have projections π_i.

This also explains why cannot be defined in terms of Δ. Even if we could apply $\Delta_{r,s}$ on the input (if the tag $r \times s$ coincided with tag $r \vee s$) we would still not be able to obtain $T^r A$ from $T^r(A \hat{\times} B)$.

This restriction is intentional – at the semantic level, it prevents manipulations with the context that would break the correspondence between tag structure and the product structure.

3.2.2 *Categorical semantics*

The categorical semantics of $\lambda_{Cs}$is shown in Figure **??**. It uses the *structural tagged comonad* structure introduced in the previous section, together with the helper operation $\Delta_{r,s}$ and the following simple helper operations:

$$\begin{aligned} \mathsf{assoc} &= \lambda(\delta_r, (\delta_s, \delta_t)) \to ((\delta_r, \delta_s), \delta_t) \\ \mathsf{swap} &= \lambda(\gamma_1, \gamma_2) \to (\gamma_2, \gamma_1) \\ f \times g &= \lambda(x, y) \to (f\ x, g\ y) \end{aligned}$$

When compared with the semantics of λ_{Cf}(Figure **??**), there is a number of notable differences. Firstly, the rule λ_{Cs} -Svar is now interpreted as ϵ_0 without the need for projection π_i. When accessing a variable, the context contains only the accessed variable. The λ_{Cs} -Sfun rule has the same structure – the only difference is that we use the $\times$ operator for combining context tags instead of $\vee$ (which is a result of the change of type signature in $_{r,s}$).

The rule λ_{Cs} -Sapp now uses the operation $_{s,(r \vee t)}$ instead of $\Delta_{s,(r \vee t)}$, which means that it splits the context instead of duplicating it. This makes the system more structural – the expressions use disjunctive parts of the context – and also explains why the composed coeffect tag is $s \times (r \vee t)$.

The only rule from λ_{Cf} that was not syntax-directed (λ_{Cf} -Ssub) is now generalized to a number of non-syntax-directed rules λ_{Cs} -SC that perform

various manipulations with the context. The semantics of $C^{r_1}\Gamma_1 \Rightarrow_c C^{r_2}\Gamma_2$ is a function that, when given a context $C^{r_1}\Gamma_1$ produces a new context $C^{r_2}\Gamma_2$. The semantics in λ_{Cs} -Sctx then takes a context, converts it to a new context which is compatible with the original expression e. The context manipulation rules work as follows:

- The λ_{Cs} -SCnest and λ_{Cs} -SCexch rules use $_{r,s}$ to split the context into a product of contexts, then perform some operation with the contexts – transform one and swap them, respectively. Finally, they re-construct a single context using $_{r,s}$.
- The λ_{Cs} -SCempty and λ_{Cs} -SCweak rules have the same semantics. They both split the context and discard one part (containing either an unused variable or an empty context).
- If we interpreted λ_{Cs} -SCcontr by applying functor $T^{r\vee s}$ to a function that duplicates a variable, the resulting context would be $C^{r\vee s}(x : \tau, x : \tau)$, which would break the correspondence between coeffect tag and context variable structure. However, that interpretation would be incorrect, because we use $\hat{\times}$ instead of normal product for variable contexts. As a result, the rule has to be interpreted as a composition of $\Delta_{r,s}$ and $_{r,s}$, which also turns a tag $r \vee s$ into $r \times s$.
- The λ_{Cs} -SCassoc rule is similar to λ_{Cs} -SCexch in the sense that it de-constructs the context, manipulates it (using assoc) and then re-constructs it.
- Finally, the λ_{Cs} -SCsub rule interprets sub-coeffecting on the context associated with a single variable using the primitive natural transformation $\iota_{r,s}$.

ALTERNATIVE: SEPARATE VARIABLES. As an alternative, we could model an expression by attaching the context separately to individual variables. This an expression $C^{r\times s}(\Gamma_1, \Gamma_2) \vdash e : \tau$ would be modelled as $C^r\Gamma_1 \times C^s\Gamma_2 \to \tau$. However, this approach largely complicates the definition of application (where tag of all variables in a context is affected). Moreover, it makes it impossible to express λ_{Cf} in terms of λ_{Cs} as discussed in Section **??**.

ALTERNATIVE: WITHOUT SUB-COEFFECTING. The semantics presented above uses the natural transformation $\iota_{r,s}$, which represents sub-coeffecting, to define the duplication operation $\Delta_{r,s}$. However, structural coeffect calculus λ_{Cs}does not require sub-coeffecting in the same way as flat λ_{Cf}(where it is required for subject reduction).

This means that it is possible to define a variant of the system that does not have the λ_{Cs} -Tsub typing rule. Then the semantics does not need the $\iota_{r,s}$ transformation, but instead, the following natural transformation has to be provided:

$$\Delta_{r,s} : T^{(r\vee s)}A \to T^rA \times T^sA$$

This variant of the system could be used to define a system that ensures that all provided context is used and is not over-approximated. This difference is similar to the difference between affine type systems (where a variable can be used at most once) and linear type systems (where a variable has to be used exactly once).

3.3 EXAMPLES OF STRUCTURAL COEFFECTS

3.3.1 *Example: Liveness analysis*

3.3.2 *Example: Data-flow (revisited)*

TODO: Also, consider additional language features that we consider for flat coeffects (mainly recursion and possibly conditionals)

3.4 CONCLUSIONS

TODO: (...)

BIBLIOGRAPHY

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

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

[3] A. Filinski. Monads in action. In *Proceedings of POPL*, 2010.

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

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

[6] 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.

[7] 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.

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

[9] 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.

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

[11] 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.

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

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

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

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

[16] 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.

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

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

[19] 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.

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

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

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

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