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 way programs execute. In recent years, this has included e. g. the development of parallel programming models (in reaction to the multi-core revolution) or improvements in data access technologies. This thesis is a response to another such revolution – the diversification of devices and systems where programs run.

The key point made by this thesis is the realization that execution environment or *context* is fundamental for writing modern applications and that programming languages should provide abstractions for programming with context and verifying how it is accessed.

We identify a number of program properties that were not connected before, but model some notion of context. Our examples include tracking different execution platforms (and their versions) in cross-platform development, 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 usage (e. g. in liveness analaysis and linear logics) or past values in stream-based data-flow programming.

Our first contribution is the discovery of the connection between the above examples and their novel presentation in the form of calculi (*coeffect systems*). The presented type systems and formal semantics highlight the relationship between different notions of context. Our second and third contributions are two unified coeffect calculi that capture commonalities in the presented examples. In particular, our *flat coeffect calculus* models languages with contextual properties of the execution environment and our *structural coeffect calculus* models languages where the contextual properties are attached to the variable usage.

Although the focus of this thesis is on the syntactic properties of the presented systems, we also discuss their category-theoretical motivation. We introduce the notion of an *indexed* comonad (based on dualisation of the well-known monad structure) and show how they provide semantics of the two coeffect calculi.

CONTENTS

1 WHY CONTEXT-AWARE PROGRAMMING MATTERS

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

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

PARALLEL PROGRAMMING. The rise of multi-core CPUs is a clear example of an external development influencing programming language research. As multi-core and multi-processor systems became ubiquitous, languages had to provide better abstractions for parallel programming. This led to the industrial popularity of *immutable* data structures (and functional programming in general), software transactional memory [8], data-parallelism and also asynchronous computing [25].

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

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

Technologies like LINQ [13] make querying data significantly easier. But perhaps because accessing data became important more gradually, it was not easy to see that SQL queries, embedded as parameterized strings[2], are a poor solution *before* better approaches were developed.

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

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

2 The dominant approach is demonstrated, for example, by a review of SQL injection prevention techniques by Clarke [3]

3 This is difficult to explain in writing and so the reader is encouraged to watch a video showing type providers for the WorldBank and CSV data sources [19].

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

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

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 in context-dependent programming are. In Chapter **??** and Chapter **??**, we develop two simple calculi to understand and capture the structure of those problems and, finally, Chapter **??** unifies the two abstractions.

1.1 WHY CONTEXT-AWARE PROGRAMMING MATTERS

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

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

- **System capabilities.** Libraries such as LINQ [13] let developers write code in a host language like C# and then cross-compile it to multiple targets (including SQL, OpenCL or JavaScript [12]). Part of the compilation (e. g. generating the SQL query) occurs at runtime and developers have no guarantee that it will succeed until the program is executed, because only subset of the host language is supported.

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

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

```
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

- **Resources & data availability.** When creating a mobile application, the program may (or may not) be granted access to device capabilities such as GPS sensor, social updates or battery status. We would like to know which of the capabilities are required and which are optional (i. e. enhance the user experience, but there is a fallback strategy). Equally, on the server-side, we might have access to different database tables 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 have to be solved. However, this is because it is not apparent that a suitable language extension could make the above problems significantly easier to solve. As the number of distinct contexts and their diversity increases, these problems will become even more commonplace.

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

1.1.1 *Context awareness #1: Platform versioning*

The diversity across devices means that developers need to target an increasing number of platforms and possibly also multiple versions of each platform. For Android, there is a number called API level [6] which "uniquely identifies the framework API revision offered by a version of the Android platform". Most changes in the libraries (but not all) are additive.

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

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

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

1.1.2 *Context awareness #2: System capabilities*

Another example related to the previous one is when libraries use metaprogramming techniques (such as LINQ [13] or F# quotations [23]) to translate code written in a subset of a host language to some other target language, such as SQL, OpenCL or JavaScript. For database access, this is a recently developed technique replacing embedded SQL discussed in the introduction, but it is a more generally important technique for programming in heterogeneous environments. It lets developers targets multiple runtimes that have limited execution capabilities.

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

```
var db = new NorthwindDataContext();

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

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

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

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

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

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

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

1.1.3 *Context awareness #3: Confidentiality and provenance*

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

For example, consider the following code sample that accesses a database by building a SQL query using string concatenation. For the purpose of the demonstration, this example does not use LINQ, but an older approach with a parameterized SQL query written as a string:

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

The code compiles without error, but it contains a major security flaw called *SQL injection* [3] (an attacker could enter `"'; DROP TABLE Products --"` as their `name` and delete the database table "Products"). For this reason, most libraries discourage building SQL commands by string concatenation, but there are still many systems that do so.

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 information about the value should be propagated to `query`. The `SqlCommand` object should then require arguments that can not directly contain user input (in an unchecked form).

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

In the security context, such marking of values (but at run-time) is called *tainting* [7], but the technique is a special case of more general *provenance* tracking. This can be useful when working with data in other contexts. For example, data jounralsts might want to propagate meta-data about the quality and the information source – is the source trustworthy? Is the data up-to-date? Such meta-data could propagate to the result and tell us important information about the calculated results.

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

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

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

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

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

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

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

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

1.2 TOWARDS CONTEXT-AWARE LANGUAGES

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

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

- **Flat coeffects** represent additional data, resources and meta-data that are available in the execution environment (regardless of how they are accessed in a program). Examples include resources such as GPS sensors and battery status (on a phone), databases (on the server), or software framework (or library) version.
- **Structural coeffects** capture additional meta-data related to inputs. This can include provenance (source of the input value), usage information (how often is the value accessed and in what ways) or security information (whether it contain sensitive data or not).

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

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

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

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

let iPhoneMain() =
  createiPhoneListing(fetchLocalNews)

let windowsMain() =
  createWindowsListing(fetchLocalNews)
```

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

1.2.1 *Context-aware languages in action*

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

We assume that the language supports cross-compilation and splits the single program into three components: one for the server-side and two for the client-side, for iPhone and Windows platforms, respectively. The cross-compilation could be done in a way similar to Links [4], but we do not require explicit annotations specifying the target platform.

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

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

1.2.2 *Understanding context with types*

The approach advocated in this thesis is to track information about context requirements using the type system. To make this practical, the system needs to provide at least partial support for automatic type inference, as the information about context requirements makes the types more complex. An inspiring example might be the F# support for units of measure [9] – the

user has to explicitly annotate constants, but the rest of the information is inferred automatically.

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

fetchLocalNews

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

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

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

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

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

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

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

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

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

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

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

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

1.3 THEORY OF CONTEXT DEPENDENCE

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

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

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

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

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

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

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

that the result is a *computation* that performs I/O effects and then returns unit value. Here, we look at another option for tracking effects, which is to extend the judgement with additional information about the effects. The judgement in a language with effect system would look as follows:

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

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

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

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

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

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

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

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

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

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

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

1.4 THESIS OUTLINE

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

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

- In Chapter **??** we give an overview of related work. Most importantly, we show that the idea of context-aware computations can be naturally approached from a number of directions developed recently in theories of programming languages (including type and effect systems, categorical semantics and sub-structural logics).

- Chapter **??** presents the first contribution of the thesis – the discovery of the connection between a number of existing programming language features that are related to context. The chapter presents type systems and semantics for a number of systems and analyses (including data-flow, liveness analysis, distributed programming and Haskell's type classes). Our novel presentation reveals their similarity.

- In Chapter **??** and Chapter **??**, we present the key contributions of this thesis. We develop the *flat* and *structural* calculi, show how they capture important contextual properties and develop their categorical semantics using a notion based on comonads. Chapter **??** links the two systems using a single formalism that is capable of capturing both flat and structural properties and also discusses an alternative presentation that is more suitable for automatic type inference of coeffects.

- Related work is presented in Chapter **??** and throughout the thesis. One important direction deserves further exploration, and so Chapter 2 starts with a brief discussion of a different approach to tracking contextual information that arises from modal logics. Finally, the rest of Chapter 2 discusses approaches for implementing the presented theory in main-stream programming languages and concludes.

If there is a one thing that the reader should remember from this thesis, it is the fact that there is a unified notion of *context*, capturing many common scenarios in programming, and that programming language designers need to provide ways for working with this context (using *coeffects* or not). This greatly reduces the number of distinct concepts that software developers need to keep in mind of when building applications for the rich and diverse execution environments of the future.

2 FURTHER WORK AND CONCLUSIONS

The main goal of this thesis is to demonstrate that there is a unified notion of *context* that captures many common scenarios in computer programming. The examples we explore include many notions of context important in modern software development (including distributed and cross-platform development) as well as context studied in fundamental programming language research (including data-flow, linear logics and variable liveness).

This thesis demonstrates that all these notions of context have something in common. In particular, they can be modelled by our flat or structural coeffect calculi. Although the results presented in this thesis are mainly of a theoretical nature, we, indeed, believe that coeffects should be integrated in main-stream programming languages.

In this chapter, we present further work in two directions. Firstly, we outline one possible approach for practical implementations of coeffects (Section 2.1). Secondly, we discuss an alternative approach to defining coeffect systems which highlights the relationship between our work and related work arising from modal logics (Section 2.2). Finally, the Section 2.5 summarizes and concludes the thesis.

2.1 TOWARDS PRACTICAL COEFFECTS

As discussed earlier, the main focus of this thesis is the development of the much needed *theory of context-aware computations* and so discussing the details of a practical implementation of coeffect tracking is beyond the scope of the thesis. However, this section briefly outlines one possible pathway towards this goal.

Many of the examples of contextual computation that we discussed earlier (e. g. implicit parameters [11] or distributed computations [15, 4]) have been implemented as a single-purpose programming language feature. However, the main contribution of this thesis is that it captures *multiple* different notions of context-aware computations using just a *single* common abstraction. For this reason, we advocate that future practical implementations of coeffects should not be single-purpose language features, but instead reusable abstractions that can be instantiated with a concrete *coeffect algebra* specified by the user.

In order to do this, programming languages need to provide two features; one that allows embedding of context-aware computations themselves in programs (akin to the "do" notation in Haskell) and one that allows tracking of the contextual information in the type system.

2.1.1 *Embedding contextual computations*

The embedding of *contextual* computations in programming languages can learn from better explored emedding of *effectful* computations. In purely functional programming languages such as Haskell, effectful computations are embedded by *implementing* them and inserting the necessary (monadic) plumbing. This is made easier by the "do" notation that inserts the monadic operations automatically.

The recently proposed "codo" notation [17] provides a similar plumbing for context-aware computations based on comonads. The notation is close to the semantics of our flat coeffect calculus (Chapter **??**). Extending it to support calculi based on the structural coeffect calculus (Chapter **??**) is an interesting futrue work – this requires explicitly manipulating individual context variables and application of structural rules (which is not needed in flat coeffects).

In ML-like languages, effects (and many coeffects) are built-in into the language semantics, but they can still use a special notation for explicitly marking effectful (coeffectful) blocks of code. In F#, this is done using *computation expressions* [20] that differ from the "do" notation in two ways. First, they support wider range of constructs, making it possible to wrap existing F# code in a block without other changes. Second, they support other abstractions including monads, applicative functors and monad transformers. It would be interesting to see if computation expressions can be extended to handle computations based on flat/structural indexed comonads.

Finally, more lightweight syntax for effectful computation can be obtained using techniques that automatically insert the necessary monadic plumbing (wihtout using any special syntax). This has been done in the context of effecful computations [22] and similar approach would be worth exploring for coeffects.

2.1.2 *Coeffect annotations as types*

The other aspect of practical implementation of coeffects is tracking the context requirements (coeffect annotations) in the type system. To achieve this, the type system needs to be able to capture the different examples of coeffect algebras. The structures used in this thesis include sets (with union or interesection), natural numbers (with addition, maximum, minimum and multiplication), two-point lattice (for liveness) and free monoids (vectors of annotations).

A recent work on embedding effect systems in Haskell [18] shows that the recent additions to the Haskell type system are powerful enough to implement structures such as sets at the type level. However, the implementation of the embedding is

[29]

[10] [24]

2.2 META-LANGUAGE

Related and further work has been discussed throughout, but one important thing remains.

Both flat coeffect calculus and structural coeffect calculus (presented in the past two chapters) use indexed comonads to define the semantics of the langauge. In this section, we follow the meta-language style and embed indexed comonads into the language – the type constructor $C^r\,\alpha$ becomes a first-class value and we add language constructs corresponding to primitive operations of the indexed comonad.

$$(\textit{var})\quad \frac{x : \alpha \in \Gamma}{\Gamma \vdash x : \alpha}$$

$$(\textit{app})\quad \frac{\Gamma \vdash e_1 : \alpha \to \beta \qquad \Gamma \vdash e_2 : \alpha}{\Gamma \vdash e_1\ e_2 : \beta}$$

$$(abs)\ \frac{\Gamma, x : \alpha \vdash e : \beta}{\Gamma \vdash \lambda x.e : \alpha \to \beta}$$

$$(letbox)\ \frac{\Gamma \vdash e_1 : C^{r \oplus s}\alpha \qquad \Gamma, x : C^{r}\alpha \vdash e_2 : \beta}{\Gamma \vdash \textbf{let box } x = e_1 \textbf{ in } e_2 : C^{s}\beta}$$

$$(eval)\ \frac{\Gamma \vdash e : C^{e}\alpha}{\Gamma \vdash !e : \alpha}$$

$$(sub)\ \frac{\Gamma \vdash e : C^{s}\alpha}{\Gamma \vdash e : C^{r}\alpha}\ (s \leqslant r)$$

2.3 RELATED WORK

This chapter is closely related to Contextual Modal Type Theory (CMTT) of Nanevski et al. However they develop their language using model logic as a basis, while we use categorical foundations as the basis - leading to a different system.

2.4 ALSO RELATED WORK

This paper follows the approaches of effect systems [5, 26, 28] and categorical semantics based on monads and comonads [14, 27]. Syntactically, *coeffects* differ from *effects* in that they model systems where λ-abstraction may split contextual requirements between the declaration-site and call-site.

Our *indexed (monoidal) comonads* (§**??**) fill the gap between (non-indexed) *(monoidal) comonads* of Uustalu and Vene [27] and indexed monads of Atkey [1], Wadler and Thiemann [28]. Interestingly, *indexed* comonads are *more general* than comonads, capturing more notions of context-dependence (§**??**).

COMONADS AND MODAL LOGICS. Bierman and de Paiva [2] model the □ modality of an intuitionistic S4 modal logic using monoidal comonads, which links our calculus to modal logics. This link can be materialized in two ways.

Pfenning et al. and Nanevski et al. derive term languages using the Curry-Howard correspondence [21, 2, 16], building a *metalanguage* (akin to Moggi's monadic metalanguage [14]) that includes □ as a type constructor. For example, in [21], the modal type □τ represents closed terms. In contrast, the *semantic* approach uses monads or comonads *only* as a semantics. This has been employed by Uustalu and Vene and (again) Moggi [14, 27]. We follow the semantic approach.

Nanevski et al. extend an S4 term language to a *contextual* modal type theory (CMTT) [16]. The *context* is a set of variables required by a computation, which makes CMTT useful for meta-programming and staged computations. Our contextual types are indexed by a coeffect algebra, which is more general and can capture variable contexts, but also integers, two-point lattices, *etc.*.

The work on CMTT suggests two extensions to coeffects. The first is developing the logical foundations. We briefly considered special cases of our system that permits local soundness in §**??** and local completeness can be treated similarly. The second problem is developing the coeffects *metalanguage*. The use of coeffect algebras would provide an additional flexibility over CMTT, allowing a wider range of applications.

2.5 SUMMARY

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

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

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

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

[7] 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.

[8] 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.

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

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

[11] 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.

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

[13] 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.

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

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

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

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

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

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

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

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

[22] 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.

[23] 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.

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

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

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

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

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

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

APPENDIX A

A.1 INTERNALIZED SUBSTITUTION

A.1.1 *First transformation*

$$(\textbf{glet}\ x = e_1\ \textbf{in}\ e_2)\ e_3 \rightsquigarrow \textbf{glet}\ x = e_1\ \textbf{in}\ (e_2\ e_3)$$

$$(app)\ \frac{(glet)\ \dfrac{\Gamma @ s \vdash e_1 : \tau_1 \qquad \Gamma, x{:}\tau_1 @ r \vdash e_2 : \tau_3 \xrightarrow{t} \tau_2}{\Gamma @ r \oplus (s \circledast r) \vdash \textbf{glet}\ x = e_1\ \textbf{in}\ e_2 : \tau_3 \xrightarrow{t} \tau_2} \qquad \Gamma @ u \vdash e_3 : \tau_3}{\Gamma @ (r \oplus (s \circledast r)) \oplus (u \circledast t) \vdash (\textbf{glet}\ x = e_1\ \textbf{in}\ e_2)\ e_3 : \tau_2}$$

to

$$(glet)\ \frac{\Gamma @ s \vdash e_1 : \tau_1 \qquad (app)\ \dfrac{\Gamma, x{:}\tau_1 @ r \vdash e_2 : \tau_3 \xrightarrow{t} \tau_2 \qquad \Gamma @ u \vdash e_3 : \tau_3}{\Gamma @ r \oplus (u \circledast t) \vdash e_2\ e_3 : \tau_2}}{\Gamma @ (r \oplus (u \circledast t)) \oplus (s \circledast (r \oplus (u \circledast t))) \vdash \textbf{glet}\ x = e_1\ \textbf{in}\ (e_2\ e_3) : \tau_2}$$

meaning

$$(r \oplus (s \circledast r)) \oplus (u \circledast t) =$$

A.1.2 *Second transformation*

Second transformation

$$(glet)\ \frac{\Gamma @ s \vdash e_s : \tau_s \qquad \Gamma, x{:}\tau_1 @ r \vdash e_r : \tau_r \qquad \Gamma, x{:}\tau_1 @ t \vdash e_t : \tau_t}{\Gamma @ t \oplus ((r \oplus (s \circledast r)) \circledast t) \vdash \textbf{glet}\ x_r = (\textbf{glet}\ x_s = e_s\ \textbf{in}\ e_r)\ \textbf{in}\ e_t : \tau_t}$$

or

$$(glet)\ \frac{\Gamma @ s \vdash e_s : \tau_s \qquad \Gamma, x{:}\tau_1 @ r \vdash e_r : \tau_r \qquad \Gamma, x{:}\tau_1 @ t \vdash e_t : \tau_t}{\Gamma @ (t \oplus (r \circledast t)) \oplus (s \circledast (t \oplus (r \circledast t))) \vdash \textbf{glet}\ x_s = e_s\ \textbf{in}\ (\textbf{glet}\ x_r = e_r\ \textbf{in}\ e_t) : \tau_t}$$

$$t \oplus ((r \oplus (s \circledast r)) \circledast t) =$$
$$t \oplus (r \circledast t) \oplus (s \circledast r \circledast t) =$$
$$s \circledast r \circledast t$$

$$(t \oplus (r \circledast t)) \oplus (s \circledast (t \oplus (r \circledast t))) =$$
$$t \oplus (r \circledast t) \oplus (s \circledast t) \oplus (s \circledast r \circledast t) =$$
$$s \circledast r \circledast t$$

require

$$r \oplus (r \circledast s) = r \circledast s$$