1 IMPLEMENTATION

In the previous three chapters, we presented the theory of coeffects consisting of type system and comonadically inspired semantics of two parameterized coeffect calculi. The theory provides a framework that simplifies the implementation of safe context-aware programming languages. To support this claim, this chapter presents a prototype implementation of three coeffect languages – language with implicit parameters and both flat and structural versions of a data-flow language.

The implementation directly follows the thoery presented in the previous three chapters. It consists of a common framework that provides type checking and translation to a simple functional target language with comonadically-inspired primitives. Each concrete context-aware language then adds a domain-specific rule for choosing unique typing derivation (as discussed in Section 4.3) together with a domain-specific definition of the comonadically-inspired primitives that define the runtime semantics (see Section 5.4).

The main goal of the implementation is to show that the theory is practically useful and to present it in a more practical way. However, we do not intend to build a complete real-world programming language. For this reason, the implementation is available primarilly as an interactive web-based essay, though it can be also downloaded and run locally.

CHAPTER STRUCTURE AND CONTRIBUTIONS

- We discuss how the implementation follows from the theory presented earlier (Section 6.1). This applies to the implementation of the *type checker* and the implementation of the *translation* to a simple target langauge that is then iterpereted. We also discuss how the common framework makes it easy to implement additional context-aware languages (Section 6.1.3).
- We consider a number of case studies (Section 6.2) that illustrate interesting aspects of the theories discussed earlier. This includes the typing of lambda abstraction and the difference between flat and structural systems (Section 6.2.1) and the comonadically-inspired translation (Section 6.2.2).
- The implementation is available not just as downloadable code, but also in the format of interactive web-based essay (Section 6.3), which aims to make coeffects accessible to a broader audience. We discuss the most interesting aspects of the essay format and briefly discuss some of the interesting implementation detail (Section 6.3.2).

1.1 FROM THEORY TO IMPLEMENTATION

The theory discussed so far provides the two key components of the implementation. In Chapter 4, we discussed the type checking of context-aware programs and Chapter 5 models the execution of context-aware programs (in terms of translation and operational semantics). For structural coeffects, the same components are discussed in Chapter **??**. In this section, we discuss how those provide foundation for the implementation.

1.1.1 *Type checking and inference*

To simplify the writing of context-aware programs, the implementation provides a limited form of type inference (Section **??**). This is available just for convenience and so we do not claim any completeness or complexity result about the algorithm and we do not present full formalization. However, it is worth noting how the domain-specific procedures for choosing a unique type derivation (Section 4.3) are adapted.

The type inference works in the standard way [31, 9] by generating type constraints and solving them. Solving of type constraints is done in the standard way, but we additionally collect and solve *coeffect constraints*. In order to obtain unique type derivation, we generate additional coeffect constraints in lambda abstraction of each flat coeffect language.

FLAT IMPLICIT PARAMETERS. As discussed in Section 4.3, when choosing unique typing derivation for implicit parameters, we keep track of the implicit parameters available in lexical scope (written as Δ). In lambda abstraction rule, the implicit parameters required by the body (tracked by r) are split so that all parameters available in lexical scope are captured and only the remaining parameters ($r \setminus \Delta$) are required from the caller of the function.

From the presentation in Section 4.3, it might appear that resolving the ambiguity related to lambda abstraction for implicit parameters requires a type system different from the core flat coeffect type system shown earlier in Section 4.2.4. This is not the case.

We track implicit parameters in scope Δ, but the rest of the (*abs*) rule from the implementation only generates an additional coeffect constraint. Writing $\Gamma \mathbin{@} t \vdash e : \tau \mid C$ for a judgement that generates coeffect constraints C, the (*abs*) rule used for implicit parameters looks as follows:

$$(abs)\ \frac{\Gamma, x{:}\tau_1; \Delta \mathbin{@} t \vdash e : \tau_2 \mid C}{\Gamma; \Delta \mathbin{@} r \vdash \lambda x{:}\tau_1 . e : \tau_1 \xrightarrow{s} \tau_2 \mid C \cup \{t = r \wedge s, r = \Delta\}}$$

Given a typing derivation for the body that produced constraints C, we generate an additional constraint that restricts r (declaration-site demands) to those available in the current static scope Δ. It is not necessary to generate a constraint for the coeffect s, because our constraint satisfaction algorithm finds the minimal set s which is $t \setminus \Delta$.

FLAT DATA-FLOW. In context-aware language for data-flow (and in language with liveness tracking), the inherent ambiguity of the (*abs*) rule is resolved by duplicating the context requirements of the body. In Section 4.3, this was defined by replacing the standard coeffect (*abs*) rule with a rule (*id-abs*) that uses an annotation r for the body of the function, declaration-site coeffect and call-site coeffect.

As with implicit parameters, the implementation does not require changing the core (*abs*) typing rule of the flat coeffect system. Instead, the unique resulution is obtained by generating additional coeffect constraints:

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

Here, the two additional constraints restrict both r and s to be equal to the coeffect of the body t and so the only possible resolution is the one specified by (*idabs*).

1.1.2 *Execution of context-aware programs*

Context-aware programs are executed by translating the source program into a simple functional target language. For simplicity, programs in the simple target language are then interpreted, but they could equally be compiled using standard techniques for compiling functional code. The translation follows the rules defined in Section 5.3 (for flat coeffect languages) and Section X (for structural coeffect languages). The result of the translation is a program that consists of the following:

- FUNCTIONAL CONSTRUCTS. Those include binary operations, tuples, let binding, constants, variables, function abstraction and application. The interpreter keeps a map of assignments for variables in scope and recursively evaluates the expression.
- COMONADIC OPERATIONS. Those are the comonadic primitives provided by indexed comonads – `cobind`, `counit` together with `merge` and `split` for flat coeffects or `merge` and `choose` for structural coeffects. The translation that produces these is shared by all context-aware languages, but their definition in the interpreter is domain-specific.
- DOMAIN-SPECIFIC OPERATIONS. Each context-aware language may additionally include operations that model domain-specific operations. For data-flow, this is `prev` (accessing past values) and for implicit parameters, this is `letimpl` and `lookup` for implicit parameter binding and access, respectively.

The fact that the prototype implementation is based on the theoretical framework provided by coeffect calculi means that it has the desirable properties proved in Section 5.4 and Section X. In particular, evaluating a well-typed context-aware program in a context that provides sufficient contextual capabilities will not cause an error.

In the interactive essay (Section 6.3), we further use the coeffects to automatically generate a user interface that requires the user to provide the required contextual capabilities (past values for individual variables, or values for implicit parameters).

Another benefit of using the common framework is that the implementation can be easily extended to support additional context-aware languages.

1.1.3 *Supporting additional context-aware languages*

The prototype implementation supports two of the context-aware languages discussed in this thesis: implicit parameters and data-flow. The remaining examples are calculus for tracking variable liveness (of ordinary variables) and the structural language based on bounded reuse (counting number of times values are used). However, the prototype implementation is based on the common coeffect framework and makes it easy to add support for these and also for other context aware languages based on coeffects.

In order to extend the implementation with support for liveness or bounded reuse tracking (or other context-aware language), the following 4 additions are required:

1. A domain-specific function abstraction rule that resolves the ambiguity in the general (*abs*) rule of the flat coeffect calculus. For liveness, the handling would be the same as for data-flow, but for other flat coeffect systems, another resolution mechanism might be used instead.

2. A domain-specific instance of coeffect algebra needs to be provided. In order to support the type inference in the prototype implementation, the constraint solver needs to be extended to solve constraint using the coeffect algebra. For liveness, this would be solving simple two-point lattice constraints.

3. For evaluation, a new kind of comonadic values needs to be added. For liveness, this would be an option value that may or may not contain a value. The semantics of comonadic operations on the values needs to be defined.

4. For context-aware languages that have additional primitives (such as `prev` or `?param`), the parser and AST needs to be extended, custom type-checking and translation rules added and domain-specific primitive operations (with their semantics) provided. Liveness and bounded reuse do not have additional custom syntax and so supporting these would not require this step.

The list mirrors a list of steps that need to be done when supporting a new effectful computation in a language that supports monadic do-notation. The step (3) corresponds to implementing a new monad and (4) corresponds to adding monad-specific effectful operations. The step (2) applies when using indexing to track effects more precisely [25]. The only step that does not have effectful/monadic counterpart is (1).

When adding a new context-aware programming language support, much of the existing infrastructure can be reused. This includes the implementation of the core coeffect and type checking rules and also the translation for standard language constructs as well as the interpreter for the target language.

1.2 CASE STUDIES

The prototype implementation illustrates a number of interesting aspects of coeffect systems. Those appear as examples in the interactive essay (discussed in Section 6.3), but we briefly review them in this section.

1.2.1 *Typing context-aware programs*

We first consider two case studies of how coeffect type checking works. The first one exposes the resolution of the ambiguity in typing for implicit parameters and the second one exposes the difference between flat and structural system for data-flow.

ABSTRACTION FOR IMPLICIT PARAMETERS. As discussed in Section 6.1.1, the implementation of the language with implicit parameters resolves the ambiguity in the lambda abstraction by generating a coeffect constraint that restricts the set of parameters required from the declaration-site to those that are lexically available. Remaining parameters are required from the call-site. This is illustrated by the following example:

```
let both =
  let ?fst = 100 in
  fun trd → ?fst + ?snd + trd in
let ?fst = 200 in
both 1
```

In this expression, the lambda function on line 3 requires implicit parameters ?fst and ?snd. Since ?fst is available in scope, the type of both is a function that requires only ?snd. In the text-based notation used in the prototype, the type of the function both is: `num -{?snd:num}-> num`.

FLAT AND STRUCTURAL DATA-FLOW. In flat data-flow, the context requirements of the body is required from both the declaration-site and from the call-site. In structural data-flow, the context requirements are tracked separately for each variable, which provides a more precise type. Consider the following two examples (the `let` keyword is used to define a curried function of two arguments):

```
let oldy x y = x + prev y in
oldy
```

When type checking the expression using the flat system, the type of oldy is inferred as `num -{1}-> num -{1}-> num`, but when using the structural system, the type becomes `num -{0}-> num -{1}-> num`.

This illustrates the difference between the two - the flat system keeps only one annotation for the whole body (which requires 1 past value). In lambda abstraction (or function declaration written using `let`), this requirement is duplicated. The structural system keeps information per-variable and so the resulting type reflects the fact that only the variable y appears inside `prev`.

1.2.2 *Comonadically-inspired translation*

In addition to running coeffect programs, the implementation can also print the result of the translation to the simple functional target language with comonadically-inspired primitives. The following two case studies illustrate important aspects of the translation for flat coeffect systems (Section 5.3) and structural coeffect systems (Section ?).

MERGING IMPLICIT PARAMETER CONTEXTS. The following example illustrates the lambda abstraction for implicit parameters. It defines a parameter ?param and then returns a function value that captures it, but also requires an implicit parameter ?other:

```
let ?param = 10 in
fun x → ?param + ?other
```

Translating the code to the target language produces the code below. The reader is encouraged to view the translation in the interactive essay (Section 6.3), which displays the types and coeffect annotations of the individual values and primitives. As in the theory, the comonadically-inspired primitives are families of operations indexed by the coeffects (we omit the annotations here):

```
let (ctx2, ctx3) = split (duplicate finput) in
let ctx1 = letimpl?param (ctx2, 10) in
fun x →
  let ctx4 = merge (x, ctx1) in
  let (ctx5, ctx6) = split (duplicate ctx4) in
  lookup?param ctx5 + lookup?other ctx6
```

The `finput` value on the first line represents an empty context in which the expression is evaluated and is of type C^{ign}unit. The context is dupli-

cated; ctx3 is not needed, because 10 is a constant; ctx2 is passed to letimpl, which assigns an implicit parameter value in the newly returned context ctx1 of type $C^{\{?param\}}$unit (this now carries the implicit parameter value, but it does not represent any ordinary variables, thus the unit type representing an empty tuple).

In the body of the function, the context ctx1 is merged with the context provided by the variable x. The type of ctx4 is $C^{\{?param,?other\}}(num \times unit)$. This is then split into two parts that contain just one of the implicit parameters and those are then accessed using lookup.

COMPOSITION IN STRUCTURAL DATA-FLOW. In structural coeffect systems, the translation works differently in that the context passed to a subexpression contains only assignments for the variables used in the subexpression (in the flat version, we always duplicated the variable context before using split). To illustrate this, consider the following simple function:

```
fun x → fun y → prev x
```

In structural coeffect systems, the comonadic value is annotated with a vector of coeffect annotations that correspond to individual variables. The initial structural input sinput is a value of type $C^{[]}()$ containing no variables (we write () rather than unit to make that more explicit). The translated code then looks as follows:

```
fun x →
  let ctx1 = merge (x, sinput) in
  (fun y →
    let ctx2 = merge (y, ctx1) in
    counit (prev (choose⟨0,1⟩ ctx2))
```

The two variables are merged with the initial context, obtaining a value ctx2 of type $C^{[0,1]}num \times num$ that contains two values with 0 and 1 past values, respectively.

For simplicity, the implementation does not use the split/merge pair of operations of the structural coeffects to obtain the correct subset of variables. This can be done, but it would make the translated code longer and more cumbersome. Instead, we use a higher-level operation choose (which can be expressed in terms of split/merge) that projects the variable subset as specified by the index. Here, $\langle 0, 1\rangle$ means that the first variable should be dropped an the second one should be kept.

The resulting single-variable context is then passed to prev (to shift the stream by one) and then to counit to obtain the current value.

1.3 INTERACTIVE ESSAY

As explained in the introduction of this chapter, the purpose of the implementation presented in this thesis is not to provide a real-world programming language, but to support the theory discussed in the rest of the thesis. The goal is to explain the theory and inspire authors of real-world programming langauges to include support for context-aware programming, ideally using coeffects as a sound foundation. For this reason, the implementation needs to be:

- ACCESSIBLE. Anyone interested should be able to use the implemented langauges without downloading the source code and compiling it and without installing specialized software.

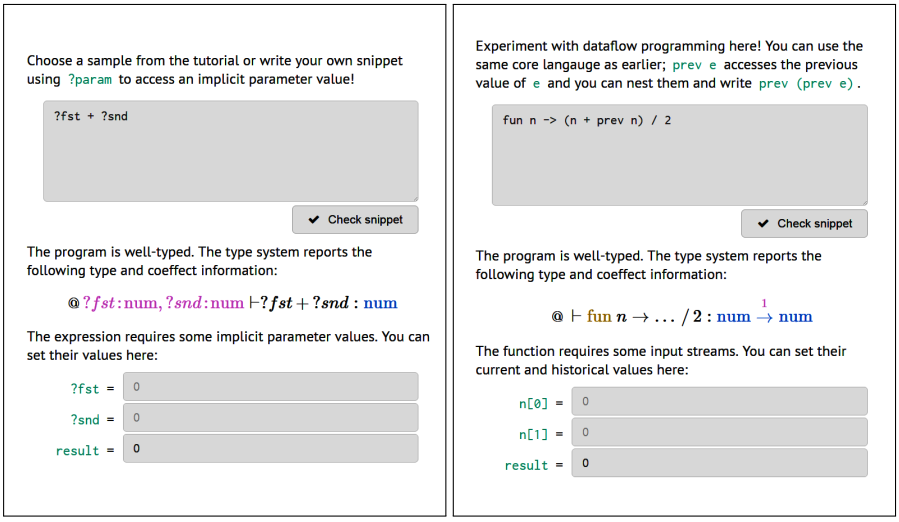

Figure 1: Interactive evaluation of implicit parameters (left) and data-flow (right)

- EXPLORABLE. It should be possible to explore the inner workings – how is the typing derived, how is the source code translated to the target language and how is it evaluated.

To make the work *accessible*, we implement sample context-aware languages in a way that makes it possible to use them in any standard web browser with JavaScript support (Section 6.3.2) without requiring any server-side component. Following the idea that "the medium is the message" [18], we choose medium that encourages *exploration* and make the implementation available not just as source code that can be compiled and run locally, but also in the format of interactive essay (Section 6.3.1). The live version of the essay can be found at: `http://tomasp.net/coeffects`.

1.3.1 *Explorable language implementation*

The interactive essay format used of the implementation is inspired by Bret Victor's work on *explorable explanations* [39]:

> *Do our reading environments encourage active reading? Or do they utterly oppose it? A typical reading tool, such as a book or website, displays the author's argument, and nothing else. The reader's line of thought remains internal and invisible, vague and speculative. We form questions, but can't answer them. We consider alternatives, but can't explore them. We question assumptions, but can't verify them. And so, in the end, we blindly trust, or blindly don't, and we miss the deep understanding that comes from dialogue and exploration.*

The interactive essay we present encourages active reading in the sense summarized in Victor's quote. We show the reader an example (program, typing derivation or translation), but the reader is encouraged to modify it and see how the explanation in the essay adapts.

The idea of active reading is older and has been encouraged in the context of art Josef Albers' classic 1963 work on color [1] (which has been turned into an interactive essay 60 years later [30]). More recently, similar formats have been used to explain topics in areas such as signal processing [32] (explaining Fourrier transformations) and sociology [11] (visualizing and

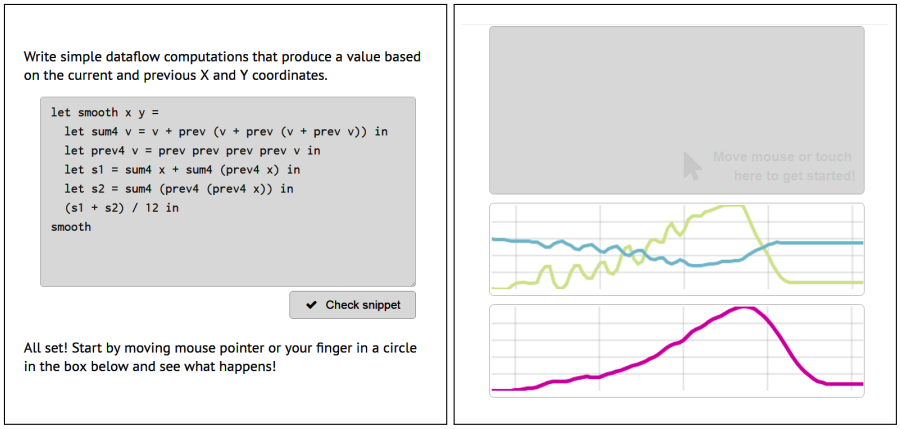

Figure 2: Function smoothing the X coordinate (left) with a sample run (right).

explaining game theoretical model of segregation in the society [33]). To our best knowledge, no such work exists in the area of programming language theory and so we briefly outline some of the interesting features that our essay provides in order to encourage active reading.

INTERACTIVE PROGRAM EXCUTION. After providing the practical motivation for coeffects (based on Chapter **??**), the interactive essay shows the reader the two sample context-aware programming languages. Readers can write code in a panel that type checks the input, generates user interface for entering the required context and runs the sample code.

The panels for implicit parameters and for data-flow computations are shown in Figure 11. The sample code on the left adds two implicit parameters and the generated UI lets the user enter implicit parameter values as required by the context in the typing judgement. The sample on the right calculates the average of the current and past value in a data-flow; the UI lets the user enter past values required by the type of the function.

The essay guides the readers through a number of interesting programs (shown in Section 6.2.1) and encourages them to run and try modifying them. For implicit parameters, this includes the case where an implicit parameter is available at both declaration-site and call-site (showing that the declaration-site value is captured). For data-flow, the examples include the comparison between the inferred type when using flat and structural type systems.

REACTIVE DATA-FLOW. The interactive program execution lets the reader run sample programs, but not in a realistic context. To show a more real-world scenario, the essay includes a widget shown in Figure 12. This lets the user write a function taking a stream of X and Y coordinates and calculate value based on the current and past values of the mouse pointer. The X and Y values, together with the result are plotted using a live chart.

In the example run shown above, the sample program calculates the average of the last 12 values of the X coordinate (green line in Figure 12). The example also illustrates one practical use of the coeffect type system – when running, the widget keeps the coordinates in a pre-allocated fixed-size array, because the coeffect type system guarantees that at most 12 past values will be accessed.

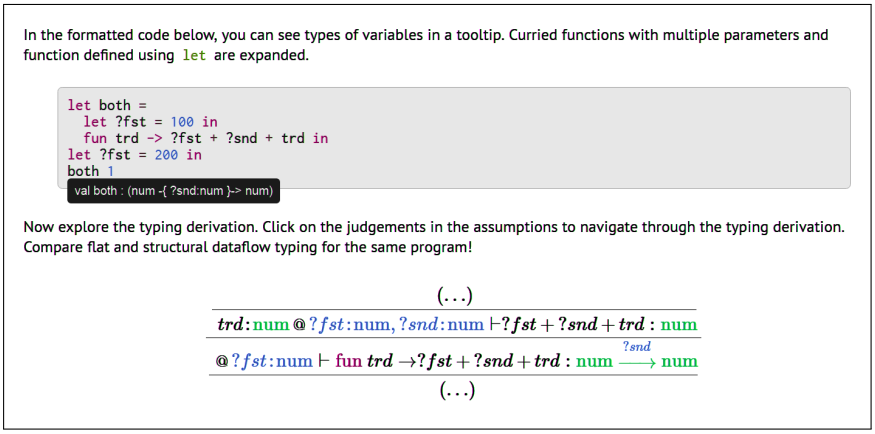

Figure 3: Source code with type information and explorable typing derivation.

EXPLORABLE TYPING DERIVATIONS. Perhaps the most important aspect of the implemented context aware programming languages is their coeffect and type system. In a conventional implementation, the functioning of the type system would remain mostly hidden – we would get an "OK" message for a well-typed program and a type error for invalid program (likely not very informative, given that reporting good error messages for type errors is a notoriously hard problem even for mature language implementations).

The presented interactive essay provides two features to help the reader understand and explore typing derivations. First, when reading code samples in the text, tooltips show the typing of individual identifiers, most interestingly functions (Figure 13, above).

Second, a later part of the essay provides a type checker that lets the user enter a source code in a context aware programming language and produces an explorable typing derivation for the program. The output (shown in Figure 13, below) displays a typing judgement with assumptions and conclusions and lets the reader navigate through the typing derivation by clicking on the assumptions or conclusions. This way, the reader can see how is the final typing derivation obtained, exploring interesting aspects, such as the abstraction rule (shown in Figure 13).

COMONADICALLY-INSPIRED TRANSLATION. The last interactive element of the essay lets the reader explore the translation of source context aware language to a target simple functional language (Section 6.1.2). Compared with the monadic do-notation [13], the comonadic translation is more complex for two reasons. First, it merges all variables into a single (comonadic) value representing the context. Second, there is a flat and structural variation of the system. For these two reasons, understanding the translation based just on the rules is harder than for monads. The essay lets readers see translation for carefuly chosen case studies that illustrate the key aspects of the translation (some discussed in Section 6.2.2), but also for their own programs encouraging *active reading*. Sample input and output are shown in Figure 14.

1.3.2 *Implementation overview*

The core part of our implementation mostly follows standard techniques for implementing type checkers and interpreters for statically-typed func-

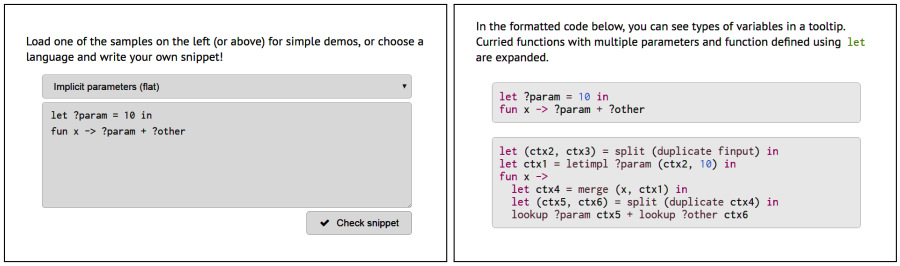

Figure 4: TBD

tional programming langauges. Two interesting aspects that are worth discussing is the JavaScript targetting (for running the langauge implementation in a web browser) and integration with client-side (JavaScript) libraries for building the user interface. The full source code can be obtained from `https://github.com/coeffects/coeffects-playground` and is structured as follows:

- Parsing is implemented using smple parser combinator library; `ast.fs` defines the abstract syntax tree for the languages; `parsec.fs` implements a small parser combinator library and `lexer.fs` with `parser.fs` parse the source code by first tokenizing the input and then parsing the stream of tokens.
- The type checking is implemented by `typechecker.fs`, which annotates an untyped AST with types and generates set of type and coeffect constraints. The constraints are later solved (using domain-specific algorithms for each of the languages) in `solver.fs`.
- Type-checked programs in context-aware languages are translated to simple target functional subset of the language in `translation.fs`; `evaluation.fs` then interprets programs in the target language. The interpretation does not handle the source language and so programs containing context-aware contstructs cannot be interpreted directly.
- The user interface of the interactive essay discussed in Section 6.3.1 is implemented partly in F# and partly in JavaScript. The two most important components include a pretty-printer `pretty.fs` which formats source code (with type tooltips) and typing derivations and `gui.fs` that implements user interaction (e.g. navigation in explorable typing derivations).

As discussed in Section 6.1.3, the implementation can be easily extended to support additional context-aware programming languages. This is due to the fact that it is based on the unified theory of coeffects. In practice, adding support for liveness tracking would require adding a domain-specific constraint solver in `solver.fs` and extending the interpreter in `evaluation.fs` with a new kind of comonadically-inspired data type (indexed maybe comonad) with its associated operations.

1.4 RELATED WORK

To our best knowledge, combining Bret Victor's idea of explorable explanations with programming language theory (as discussed in Section 6.3) is a novel contribution of our work. On the technical side, we build on a number of standard methods.

PARSING AND TYPE CHECKING. The implementation of the parser, type checker and interpreter follows standard techniques for implementing functional programming languages. In order to be able to compile the implementation to JavaScript (see below), we built a small parser combinator library [12] rather than using one of the already available libraries [37].

TARGETTING JAVASCRIPT. In order to make the implementation accessible to a broad audience, it can be executed in a web browser. This is achieved by automatically translating the implementation from F# to JavaScript. We use an F# library called FunScript [5] (which is a more recent incarnation of the idea developed by the author [26]). We choose F#, but similar tools exist for other functional languages such as OCaml [40]. It is worth noting that FunScript is implemented as a *library* rather than as a compiler extension. This is done using the meta-programming capabilities of F# [34].

CLIENT-SIDE LIBRARY INTEGRATION. An interesting aspect of the interactive essay user interface is the integration with third-party JavaScript libraries. We use a number of libraries including JQuery (for web browser DOM manipulation) and MathJax [7] (for rendering of typing derivation). In order to call those from F# code, we use a number of mapping methods described in [27]. For example, the following F# declaration is used to invoke the Queue function in MathJax:

```
[⟨JSEmitInline("MathJax.Hub.Queue({0});")⟩]
let queueAction (f : unit → unit) : unit = failwith "JS only!"
```

The JSEmitInline syntax on the first line is an attribute that instructs FunScript to compile all invocations of the queueAction function into the JavaScript literal specified in the attribute (with `{0}` replaced by the argument).

1.5 SUMMARY

This chapter supplements the theory of coeffects presented in the previous three chapters with a prototype implementation. We implemented three simple context-aware programming langauges that track implicit parameters and past values in dataflow computations (in flat and structural way).

The implementation discussed in this chapter provides an evidence for our claim that the theory of coeffects can be used as a basis for a wide range of sound context-aware programming languages. Our implementation consists of a shared coeffect framework (handling type checking and translation). Each context-aware language then adds a domain-specific rule for choosing a unique typing derivation, an interpretation of comonadically-inspired primitives and (optionally) domain-specific primitives such as the `prev` construct for dataflow.

We make the implementation available in the usual form (as source code that can be downloaded, compiled and executed), but we also present it in the form of interactive essay. This encourages *active reading* and lets the reader explore a number of aspects of the implementation including the type checking (through explorable typing derivations) and the translation. The key contribution of this thesis is that it provides a unified way for *thinking* about context in programming langauges and the interactive presentation of the implementation is aligned with this goal. Programming languages of the future will need a mechanism akin to coeffects and we aim to provide a convincing argument supported by a prototype implementation.

BIBLIOGRAPHY

[1] J. Albers. *Interaction of color*. Yale University Press, 2013.

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

[3] A. Bove, P. Dybjer, and U. Norell. A brief overview of agda–a functional language with dependent types. In *Theorem Proving in Higher Order Logics*, pages 73–78. Springer, 2009.

[4] E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(05):552–593, 2013.

[5] Z. Bray. Funscript: F# to javascript with type providers. Available at http://funscript.info/, 2016.

[6] A. Brunel, M. Gaboardi, D. Mazza, and S. Zdancewic. A core quantitative coeffect calculus. In *ESOP*, pages 351–370, 2014.

[7] D. Cervone. Mathjax: a platform for mathematics on the web. *Notices of the AMS*, 59(2):312–316, 2012.

[8] M. M. Chakravarty, G. Keller, and S. P. Jones. Associated type synonyms. In *ACM SIGPLAN Notices*, volume 40, pages 241–253. ACM, 2005.

[9] L. Damas. Type assignment in programming languages. 1984.

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

[11] V. Hart and N. Case. Prable of the polygons: A playable post on the shape of society. Available at http://ncase.me/polygons/, 2014.

[12] G. Hutton and E. Meijer. Monadic parser combinators. 1996.

[13] S. L. P. Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[14] S.-y. Katsumata. Parametric effect monads and semantics of effect systems. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 633–645, New York, NY, USA, 2014. ACM.

[15] 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.

[16] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 47–57, New York, NY, USA, 1988. ACM.

[17] C. McBride. Faking it simulating dependent types in haskell. *Journal of functional programming*, 12(4-5):375–392, 2002.

[18] M. McLuhan and Q. Fiore. The medium is the message. *New York*, 123:126–128, 1967.

[19] R. Milner. *The Definition of Standard ML: Revised*. MIT press, 1997.

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

[21] D. L. Niki Vazou. Remarrying effects and monads. *Proceedings of MSFP (to appear)*, 2014.

[22] D. Orchard. Programming contextual computations.

[23] D. Orchard. Should I use a Monad or a Comonad? Unpublished draft, 2012.

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

[25] 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.

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

[27] T. Petricek, D. Syme, and Z. Bray. In the age of web: Typed functional-first programming revisited. In *Post-proceedings of ML Workshop*, ML 2014.

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

[29] B. C. Pierce. *Types and programming languages*. MIT press, 2002.

[30] Potion Design Studio, based on the work of Josef Albers. Interaction of color: App for iPad. Available at `http://yupnet.org/interactionofcolor/`, 2013.

[31] F. Pottier and D. Rémy. The essence of ml type inference, 2005.

[32] J. Schaedler. Seeing circles, sines, and signals: A compact primer on digital signal processing. Available at `https://github.com/jackschaedler/circles-sines-signals`, 2015.

[33] T. Schelling. Dynamic models of segregation. *Journal of mathematical sociology*, 1(2):143–186, 1971.

[34] 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.

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

[36] 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.

[37] S. Tolksdorf. Fparsec-a parser combinator library for f#. Available at http://www.quanttec.com/fparsec, 2013.

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

[39] B. Victor. Explorable explanations. Available at http://worrydream.com/ExplorableExplanations/, 2011.

[40] J. Vouillon and V. Balat. From bytecode to javascript: the js_of_ocaml compiler. *Software: Practice and Experience*, 44(8):951–972, 2014.

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

[42] A. K. Wright and M. Felleisen. A Syntactic Approach to Type Soundness. *Information and computation*, 115(1):38–94, 1994.

[43] B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation*, pages 53–66. ACM, 2012.