

# Elaine: Elaborations of Higher-Order Effects as First-Class Language Feature

---

*Version of June 23, 2023*

Terts Diepraam



---

# Elaine: Elaborations of Higher-Order Effects as First-Class Language Feature

---

THESIS

submitted in partial fulfillment of the  
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Terts Diepraam  
born in Amsterdam, the Netherlands



Programming Languages Group  
Department of Software Technology  
Faculty EEMCS, Delft University of Technology  
Delft, the Netherlands  
[www.ewi.tudelft.nl](http://www.ewi.tudelft.nl)

© 2023 Terts Diepraam.

Cover picture: Random maze.

---

# Elaine: Elaborations of Higher-Order Effects as First-Class Language Feature

---

Author: Terts Diepraam  
Student id: 5652235  
Email: t.diepraam@student.tudelft.nl

## Abstract

### Thesis Committee:

Chair:	Prof. dr. C. Hair, Faculty EEMCS, TU Delft
Committee Member:	Dr. A. Bee, Faculty EEMCS, TU Delft
Committee Member:	Dr. C. Dee, Faculty EEMCS, TU Delft
University Supervisor:	Ir. E. Ef, Faculty EEMCS, TU Delft



---

# Preface

Preface here.

Terts Diepraam  
Delft, the Netherlands  
June 23, 2023





---

# Contents

<b>Preface</b>	<b>iii</b>
<b>Contents</b>	<b>v</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Contributions . . . . .	2
<b>2 Background</b>	<b>3</b>
2.1 Monads and Monad Transformers . . . . .	3
2.2 Algebraic Effects . . . . .	4
2.3 Elaborations . . . . .	7
<b>3 A Tour of Elaine</b>	<b>9</b>
3.1 Basics . . . . .	9
3.2 Algebraic Effects . . . . .	10
3.3 Higher-Order Effects in Elaine . . . . .	13
<b>4 Elaine Specification</b>	<b>17</b>
4.1 Syntax definition . . . . .	17
4.2 Typing judgments . . . . .	20
4.3 Desugaring . . . . .	23
4.4 Elaboration resolution . . . . .	23
4.5 Semantics . . . . .	23
<b>5 Implicit Elaboration Resolution</b>	<b>25</b>
<b>6 Elaboration Compilation</b>	<b>27</b>
6.1 Observations about Elaborations . . . . .	27
6.2 Encoding Macros as Functions . . . . .	28
6.3 Handlers as Dictionary Passing . . . . .	29
6.4 Compiling Elaborations into Handlers . . . . .	29
6.5 An Alternative Design for Elaine . . . . .	30
<b>7 Related Work</b>	<b>33</b>
<b>8 Conclusion</b>	<b>37</b>
<b>Bibliography</b>	<b>39</b>



# Chapter 1

---

## Introduction

As a program runs, it usually interacts with its environment. A function might, for example, allocate some memory, open a file or throw an exception. Apart from producing a value, such a procedure therefore also has some other observable *effects* (**moggi\_computational\_1989**).

The classical  $\lambda$ -calculus does not take these effects into account and models only *pure* computations (**moggi\_computational\_1989**). That is, computations which take some input and for every set of inputs they return the same output, without interacting with their environment. There is an elegance to this view, since pure programs are simple to reason about and analyse.

However, in practice, many programs are *impure* or *effectful*. As **moggi\_computational\_1989** notes, analysing only pure computation leaves out many aspects of programs, such as side effects, non-determinism and non-termination. Reasoning about the full behaviour of a program then necessarily needs to include an analysis of effects.

Some programming languages, such as C, give the programmer virtually unrestricted access to effectful operations. Any part of a C program can access memory and the filesystem and is even allowed to use `goto` to jump to any location in the code outside the structured control flow constructs. This unrestricted use has famously been criticized by **dijkstra\_letters\_1968**. In C (and similar languages), effects are also largely ignored in the type system. Programmers therefore have to rely on documentation and the source code to determine the side effects of any given function.

A challenge in language design is then to design a language which allows for (limited) impure calculation, while retaining the attractive qualities of pure languages. As a result, programming languages often have dedicated features for some effects, such as exceptions, coroutines and generators. While these features might seem very different on the surface, they have an important common property: they give away control to some other procedure and are handed back control later. This procedure could be a memory allocator, a scheduler, the kernel, an exception handler or something else that implements the necessary operations to perform the effect.

This insight is at the code of the theory of *algebraic effects*, which unifies many effects into a single concept (**goos\_adequacy\_2001**; **castagna\_handlers\_2009**). Here, effectful computations can only be used within *effect handlers*, which the computation yields control to. A handler can then in turn call the continuation of the computation.

In recent years, some languages (e.g. Koka (**leijen\_koka\_2014**), Eff (**bauer\_programming\_2015**), Frank (**lindley\_be\_2017**), and Effekt (**brachthauser\_effects\_2020**)) have been created with support for algebraic effects. These languages allow the programmer to define new effects without needing a dedicated language feature.

Moreover, these languages often feature type systems that can reason about the effects in each function. The programmer can therefore see from the signature what a function can do, such that effects act as capabilities (**brachthauser\_effects\_2020**). Conversely, a library

author might choose to only disallow (some) effects in some functions. For example, a hash function is generally understood to be deterministic and effectless, because it needs to be reproducible.

However, there are some commonly used effects that are not algebraic, namely *higher-order effects*. That is, effects whose operations take effectful computations as arguments that do not behave as continuations. To overcome this limitation, **bach\_poulsen\_hefty\_2023** have extended algebraic effects with *hefty algebras*. To support higher-order effects, they introduced *effect elaborations* in addition to handlers.

In this thesis, we introduce a novel programming language called *Elaine*. Like Koka and other languages with support for algebraic effects, Elaine supports effect rows and handlers. Unlike those languages, Elaine additionally also supports elaborations and higher-order effects. We define the syntax, reduction semantics and type system of Elaine and provide a full implementation of the language in Haskell<sup>1</sup>. This implementation includes a parser, type checker, pretty printer and type checker. Additionally, we introduce a novel feature that allows elaborations to be inferred to reduce the syntactic overhead of elaborations.

Finally, we give a transformation from higher-order effects to algebraic effects. **This transformation shows that elaborations can be added to existing languages and libraries for effects with relative ease.**

With Elaine, we argue that elaborations are a natural and easy representation of higher-order effects with a set of example programs.

## 1.1 Contributions

The main contribution of this thesis is the specification and implementation of Elaine. This consists of several parts.

- We define a syntax suitable for a language with both handlers and elaboration.
- We introduce a type inference system for a language with higher-order effects and elaborations.
- We present a novel representation of effect rows as multiset which, though semantically equivalent to earlier representations, allows for a simple definition of effect row unification.
- We propose that elaborations should be inferred in most cases and provide a type-directed scheme for this inference in Chapter 5.
- We define 2 procedures for transforming programs with elaborations and higher-order effects to programs with only handlers and algebraic effects.

---

<sup>1</sup>Available at <https://github.com/tertsdiepraam/thesis/tree/main/elaine>

## Chapter 2

---

# Background

While hefty algebras are a relatively new concept, the analysis of effectful computation and the representation of effects in programming language has a rich history. This history is relevant to this thesis because the various approaches to modelling effects proposed over time can be found in many popular programming languages. A comparison between these languages and languages with algebraic and higher-order effects hence requires comparison between theories. This chapter details the parts of this history that are relevant to Elaine.

### 2.1 Monads and Monad Transformers

The study of effects starts right at the two foundational theories of computation:  $\lambda$ -calculus and Turing machines. Their respective treatment of effects could not be more different. The former is only concerned with pure computation, while the latter consists solely of effectful operations.

In  $\lambda$ -calculus, effects are not modelled; every function is a function in the mathematical sense, that is, a pure computation (**moggi\_computational\_1989**). Hence, many observable properties of programs are ignored, such as non-determinism and side effects. In their seminal paper, **moggi\_computational\_1989** unified *monads* with computational effects, which they initially called notions of computation. **moggi\_computational\_1989** identified that for any monad  $T : C \rightarrow C$  and a type of values  $A$ , the type  $TA$  is the type of a computation of values of type  $A$ .

Since many programming languages have the ability to express monads from within the language, monads became a popular way to model effectful computation in functional programming languages. In particular, **peyton\_jones\_imperative\_1993** introduced a technique to model effects via monads in Haskell. This technique keeps the computation pure, while not requiring any extensions to the type system.

A limitation of treating effects as monads is that they do not compose well; the composition of two monads is not itself a monad. A solution to this are *monad transformers*, which are functors over monads that add operations to a monad (**moggi\_abstract\_1989**). A regular monad can then be obtained by applying a monad transformer to the `Identity` monad. The representation of a monad then becomes much like that of a list of monad transformers, with the `Identity` monad as `Nil` value. This “list” of transformers is ordered. For example, using the terminology from Haskell’s `mtl` library, the monad `StateT a (ReaderT b Identity)` is distinct from `ReaderT b (StateT a Identity)`. The order of the monad transformers also determines the order in which they must be handled: the outermost monad transformer must be handled first.

In practice, this model has turned out to work quite well, especially in combination with `do`-notation, which allowed for easier sequential execution of effectful computations.

## 2.2 Algebraic Effects

**TODO:** Introduce algebraic effects, effect rows, and handlers.

### 2.2.1 Algebraic Theories

This section introduces algebraic theories. In particular, we will discuss algebraic theories with parametrized operations and general arities. This will form a foundation on which we can define algebraic effects. The definitions in this section follow [bauer\\_\\_what\\_\\_2018](#).

**TODO:** This section is *probably* too long and goes too much into the theory of algebraic theories

**Definition 1** (Signature). A *signature*  $\Sigma = \{(op_i, P_i, A_i)\}$  is a collection of operation symbols  $op_i$  with corresponding parameter sets  $P_i$  and arity sets  $A_i$ . We will write operation symbols as follows:

$$op_i : P_i \rightsquigarrow A_i.$$

The arities are arbitrary sets. However, using the von Neumann ordinals, we can use natural numbers as arities. Hence, we can call operation symbols with arities 1, 2 and 3 *unary*, *binary* and *ternary* respectively. An operation symbol with arity 0 is then called *constant* or *nullary*. Two other common arities are  $\emptyset = \{\}$  and  $\mathbf{1} = \{()\}$ . We will refer to  $()$  as the unit.

We can build terms with any given signature by composing the operations. Given some set  $X$ , we can build a set  $\text{Tree}_\Sigma(X)$  of *well-founded trees* over  $\Sigma$  generated by  $X$ . This set is defined inductively:

- for every  $x \in X$  we have a tree **return**  $x$ ,
- and if  $p \in P_i$  and  $\kappa : A_i \rightarrow \text{Tree}_\Sigma(X)$  then  $op_i(p, \kappa)$  is a tree, where  $op_i$  is the label for the root and the subtrees are given by  $\kappa$ .

A  $\Sigma$ -term is a pair of a context  $X$  and a tree  $t \in \text{Tree}_\Sigma(X)$ . We write a  $\Sigma$ -term as

$$X \mid t.$$

While the notation for these trees is intentionally evocative of functions in many programming languages, it is important to note that the terms are only a representation of a tree and should be thought of as such.

**Definition 2** ( $\Sigma$ -equation). A  $\Sigma$ -equation is a pair of  $\Sigma$ -terms and a context  $X$ . We denote the equation

$$X \mid l = r.$$

Here, the  $=$  symbol is just notation and its meaning is left unspecified. Note that we can only create equations with the  $=$  symbol. We cannot, for instance, create an equation  $X \mid l \neq r$ .

When the relevant signature  $\Sigma$  is unambiguous, we will omit the  $\Sigma$  from the definitions above and simply speak of terms and equations. We can now build terms and equations with any signature we define. Hence, we can give a signature along with some associated laws that we intend to hold for that signature; this is the idea of an algebraic theory.

**Definition 3** (Algebraic theory). An *algebraic theory* (or *equational theory*) is a pair  $\mathsf{T} = (\Sigma_{\mathsf{T}}, \mathcal{E}_{\mathsf{T}})$  consisting of a signature  $\Sigma_{\mathsf{T}}$  and a collection  $\mathcal{E}_{\mathsf{T}}$  of  $\Sigma_{\mathsf{T}}$ -equations.

An algebraic theory is still hollow; it is only a specification, not an implementation. The implementation or meaning of the operations that we apply to the operation symbols needs to be given via an interpretation.

**Definition 4** (Interpretation). An *interpretation*  $I$  of a signature  $\Sigma$  is

1. a *carrier set*  $|I|$
2. and for each operation  $op_i$  a map

$$\llbracket op_i \rrbracket_I : P_i \times |I|_i^A \rightarrow |I|,$$

called an *operation*.

Additionally, we can define the interpretation of a tree, and by extension of a term, as a map

$$\llbracket t \rrbracket_I : |I|^X \rightarrow |I|.$$

This map is defined as

$$\llbracket \text{return } x \rrbracket_I : \eta \mapsto \eta(x) \quad \llbracket op_i(p, \kappa) \rrbracket_I : \eta \mapsto \llbracket op_i \rrbracket_I(p, \lambda a. \llbracket \kappa(a) \rrbracket_I(\eta)).$$

If the choice of  $I$  is obvious from the context, we will omit the subscript.

The *semantic bracket*  $\llbracket \cdot \rrbracket_I$  is used to indicate that syntactic constructs are mapped to some (mathematical) interpretation of those symbols. In other words, an interpretation gives denotational semantics to a signature.

**QUESTION:**  
this true?

**Definition 5** (Model). We say that a  $\Sigma$ -equation  $X \mid l = r$  is *valid*, if the interpretations of  $l$  and  $r$  evaluate to the same map, that is,

$$\llbracket l \rrbracket = \llbracket r \rrbracket.$$

A  $\mathsf{T}$ -*model*  $M$  of an algebraic theory  $\mathsf{T}$  is an interpretation of  $\Sigma_{\mathsf{T}}$  for which all the equations in  $\mathcal{E}_{\mathsf{T}}$  valid.

We can relate models to each other with morphisms between their carrier sets. Given models  $L$  and  $M$  for  $\mathsf{T}$ , we call such a morphism  $\phi : |L| \rightarrow |M|$ , a  $\mathsf{T}$ -homomorphism if the following condition holds for all  $op_i$  in  $\Sigma_{\mathsf{T}}$ :

$$\phi \circ \llbracket op_i \rrbracket_L(p, \kappa) = \llbracket op_i \rrbracket_M(k, \phi \circ k).$$

That is, if  $\phi$  commutes with operations. The models and the  $\mathsf{T}$ -homomorphisms for a category  $\mathbf{Mod}(\mathsf{T})$  acting as the objects and morphisms, respectively.

Crucially, we can create a *free model* for each algebraic theory, which is an initial object in  $\mathbf{Mod}(\mathsf{T})$ . The free model consists of the trees and equivalence relations between the trees.

### 2.2.2 Notation for Computations

To reason about computations, we need to introduce some notation.

A computation can either be pure or effectful. A pure computation only returns a value, while an effectful computation performs some operation and then continues. We write  $op(p_1, \dots, p_n)$  for some (effectful) operation  $op$ , with parameters  $p_1, \dots, p_n$ .

We can sequence operations with a syntax reminiscent of do-notation:

$$\{x \leftarrow op(p_1, \dots, p_n); \kappa'\}.$$

A sequence will be surrounded by  $\{\}$  when the notation is otherwise ambiguous. When a value from an operation in a sequence is discarded, we will omit the variable assignment and write

$$\{op(p_1, \dots, p_n); \kappa\}.$$

Additionally, we will define a *bind* operation  $\gg=$  as follows, where  $x$  is some fresh variable:

$$\kappa \gg= \kappa' \stackrel{\text{def}}{=} \{x \leftarrow \kappa; \kappa'(x)\}.$$

As an example, take the **State** effect. To use this effect, we need two operations: **put** and **get**. The computation

$$\{\text{put}(a); \text{get}()\}$$

then first performs the **put** and then the **get** operation, returning the result of the **get** operation.

### 2.2.3 Effects as Algebraic Theories

**goos\_adequacy\_2001** have shown that many effects can be represented as algebraic theories. Naturally, this representation of computation matches the definition of trees given above. Hence, we can connect the dots. We can represent computations as terms, so what are the signatures and equations? We start with the signature for **State**:

$$\text{put} : S \rightsquigarrow 1 \quad \text{and} \quad \text{get} : 1 \rightsquigarrow S.$$

This signature indicates that **put** takes an  $S$  as parameter and resumes with  $()$  and that **get** takes  $()$  and resumes with  $S$ . Now we can define the equations that we want **State** to follow:

$$\begin{aligned} s \leftarrow \text{get}(); t \leftarrow \text{get}(); \kappa(s, t) &= s \leftarrow \text{get}(); \kappa(s, s) \\ \text{get}() \gg= \text{put}; \kappa() &= \kappa() \\ \text{put}(s); \text{get}() \gg= \kappa &= \text{put}(s); \kappa(s) \\ \text{put}(s); \text{put}(t); \kappa() &= \text{put}(t); \kappa() \end{aligned}$$

This gives us an algebraic theory corresponding to the **State** monad. **goos\_adequacy\_2001** have shown that this theory gives rise to the canonical **State** monad. Many other effects can also be represented as algebraic theories, including but not limited to, non-determinism, non-termination, iteration, cooperative asynchronicity, traversal, input and output [citation needed](#). These effects are called *algebraic effects*.

As shown by **plotkin\_algebraic\_2003**, an effect is algebraic if and only if it satisfies the *algebraicity property*, which can be expressed as follows:

$$op(m_1, \dots, m_n) \gg= \kappa = op(p, m_1 \gg= \kappa, \dots, m_n \gg= \kappa).$$

In other words, all computation parameters must be *continuation-like*, that is, they are some computation followed by the continuation (**bach\_poulsen\_hefty\_2023**). This property is called the *algebraicity property* (**plotkin\_algebraic\_2003**).

A simple effect for which the algebraicity property does not hold is the **Reader** monad with the **local** and **ask** operations. The intended effect is that **local** applies some transformation  $f$  to the value retrieved with **ask** within the computation  $m$ , but not outside  $m$ . Therefore, we have

$$\text{local}(f, m) \gg= \text{ask}() \neq \text{local}(f, m \gg= \text{ask}()),$$

and have to conclude that we cannot represent the **Reader** monad as an algebraic theory and the effect is not algebraic.

A similar argument goes for the **Exception** effect. The **catch** operation takes two computation parameters, it executes the first and jumps to the second on encountering the **throw** operation. The problem arises when we bind with an **throw** operation:

$$\text{catch}(m_1, m_2) \gg= \text{throw}() \neq \text{catch}(m_1 \gg= \text{throw}(), m_2 \gg= \text{throw}()).$$

On the left-hand side,  $m_2$  will not be executed if  $m_1$  does not throw, while on the right-hand side,  $m_2$  will always get executed. This does not match the semantics we expect from the **catch** operation.



### 2.2.4 Effect Handlers

The distinction between effects which are and which are not algebraic has been described as the difference between *effect constructors* and *effect destructors* ([plotkin\\_algebraic\\_2003](#)). The `local` and `catch` operations have to act on effectful computations and change the meaning of the effects in that computation. So, they have to deconstruct the effects in their computations.

[castagna\\_handlers\\_2009](#) introduced *effect handlers* as a mechanism to allow for this deconstruction. Effect handlers are a generalization of exception handlers. They define the implementation for a set of algebraic operations in the sub-expression.

For example, we can define a handler for just the `ask` operation, which is algebraic:

$$hAsk(x) = \text{handler} \{ \text{return}(x) \mapsto x, \\ \text{ask}() \ \kappa \mapsto \kappa(x) \}.$$

The `handle` construct then applies a handler to an expression. For instance, the following computation with `return` with the value 5:

$$\text{handle}[hAsk(5)] \ \text{ask}().$$

With that handler we can give a definition of `local` that has the intended behaviour:

$$\text{local}(f, m) \stackrel{\text{def}}{=} \{x \leftarrow \text{ask}(); \text{handle}[hAsk(f(x))] \ m\}.$$

However, `local` cannot be defined as an algebraic operation, meaning that we cannot write a handler for it, it can only be defined as a handler. This is known as the *modularity problem* with higher-order effects ([wu\\_effect\\_2014](#)).

## 2.3 Elaborations

Several solutions to the modularity problem have been proposed ([wu\\_effect\\_2014](#); [oh\\_latent\\_2021](#)). Most recently, [bach\\_poulsen\\_hefty\\_2023](#) introduced hefty algebras. The idea behind hefty algebras is that an additional layer of modularity is introduced, specifically for higher-order effects. The higher-order operations are not algebraic, but they can be *elaborated* into algebraic operations.

A computation with higher-order effects is then first elaborated into a computation with only algebraic effects. The remaining algebraic effects can then in turn be handled to yield the result of the computation.

The advantage of hefty algebras over previous approaches is that the elaboration step is quite simple and that the result is a computation with regular algebraic effects.

Continuing the `local` example, we can make an elaboration based on the definition above:

$$eLocal \stackrel{\text{def}}{=} \text{elaboration} \{ \\ \text{local}!(f, m) \mapsto \{v \leftarrow \text{ask}(); \text{handle}[hAsk(f(v))] \ m\} \\ \},$$

We can then apply this elaboration to an expression with the `elab` keyword, similarly to `handle`:

$$\text{handle}[hAsk(5)] \ \text{elab}[eLocal] \{ \\ \quad x \leftarrow \text{ask}(); \\ \quad y \leftarrow \text{local}!(\lambda x. 2 \cdot x, \{\text{ask}()\}); \\ \quad x + y \\ \}$$

After the elaboration step, the computation will be elaborated into the program below, which will evaluate to 15.

```

handle[hAsk(5)] {
  x ← ask();
  y ← {
    v ← ask();
    handle[hAsk((λx. 2 · x)(v))] ask()
  };
  x + y
}

```

One way to think about elaboration operations is as scoped modular macros; a syntactic substitution is performed based on the given elaboration.

Throughout this thesis we will write elaborated higher-order operations with a ! suffix, to distinguish them from algebraic effects.

## Chapter 3

---

# A Tour of Elaine

The language designed for this thesis is called “Elaine”. The distinguishing feature of this language is its support for higher-order effects via elaborations. As far as we know, it is the second language with support for elaborations, the first being Heft. Elaine adds two new features: implicit elaboration resolution and compilation of elaborations, which are explained in Chapters 5 and 6, respectively. Additionally, Elaine differs by not requiring monadic style programming for effectful computation. This makes Elaine a surprisingly expressive language given its simplicity.

This chapter introduces Elaine with motivating examples for the design choices. The full specification is given in Chapter 4. More example programs are available online<sup>1</sup>.

### 3.1 Basics

The design of Elaine is similar to Koka, with syntactical elements inspired by Rust. Apart from the elaborations and handlers, the language should not be particularly surprising; it has standard `let` bindings, if-else expressions, first-class functions, booleans, integers and strings.

An Elaine program consists of a tree of modules. Top level declarations are part of the root module. The result of the program will be the value assigned to the `main` variable in the root module. A module is declared with **mod**, which takes a name and a block of declarations. Declarations can be marked as public with the **pub** keyword. A module’s public declarations can be imported into another module with **use**.

The built-in primitives are `Int`, `Bool`, `String` and the unit `()`. The `std` module provides functions for basic manipulation of these primitives (e.g. `mul`, `lt` and `sub`). Functions are defined with **fn**, followed by a list of arguments and a function body. Functions are called with parentheses.

The type system features Hindley-Milner style type inference. Let bindings, function arguments and function return types can be given explicit types. By convention, we will write variables and modules in lowercase and capitalize types.

The language does not support recursion or any other looping construct.

Below is a program that prints whether the square of 4 is even or odd.

```
1 # The standard library contains basic functions for manipulation
2 # of integers, booleans and strings.
3 use std;
4
5 # Functions are created with `fn` and bound with `let`, just like
6 # other values. The last expression in a function is returned.
7 let square = fn(x: Int) Int {
```

---

<sup>1</sup><https://github.com/tertsdiepraam/thesis/tree/main/elaine/examples>

```
8   mul(x, x)
9 };
10
11 let is_even = fn(x: Int) Bool {
12   eq(0, modulo(x, 2))
13 };
14
15 # Type annotations can be inferred:
16 let square_is_even = fn(x) {
17   let result = is_even(square(x));
18   if result { "even" } else { "odd" }
19 };
20
21 let give_answer = fn(f, x) {
22   let prefix = concat(concat(s, " "), show_int(x));
23   let text = concat(prefix, " is ");
24   let answer = f(x);
25   concat(text, answer)
26 };
27
28 let main = give_answer(square_is_even, 4);
```

## 3.2 Algebraic Effects

The programs in the previous section are all pure and contain no effects. Like the languages discussed in ??, Elaine additionally has first class support for effects and effect handlers.

An effect is declared with the **effect** keyword. An effect needs a name and a set of operations. Operations are the functions that are associated with the effect. They can have an arbitrary number of arguments and a return type. Only the signature of operations can be given in an effect declaration, the implementation must be provided via handlers (see Section 3.2.2).

### 3.2.1 Effect Rows

**TODO:** Contextual vs parametric effect rows (see effects as capabilities paper). The paper fails to really connect the two: contextual is just parametric with implicit variables. However, it might be more convenient. The main difference is in the interpretation of purity (real vs contextual). In general, I'd like to have a full section on effect row semantics. In the capabilities paper effect rows are sets, which makes it possible to do stuff like (leijen\_extensible\_2005).

In Elaine, each type has an *effect row*. In the previous examples, this effect row has been elided, but it is still inferred by the type checker. Effect rows specify the effects that need be handled to within the expression. For simple values, that effect row is empty, denoted  $\langle \rangle$ . For example, an integer has type  $\langle \rangle$  Int. Without row elision, the `square` function in the previous section could therefore have been written as

```
1 let square = fn(x: <> Int) <> Int {
2   mul(x, x)
3 }
```

Simple effect rows consist of a list of effect names separated by commas. The return type of a function that returns an integer and uses effects "A" and "B" has type  $\langle A, B \rangle$  Int. Important here is that this type is equivalent to  $\langle B, A \rangle$  Int: the order of effects in effect rows is irrelevant. However, the multiplicity is important, that is, the effect rows  $\langle A, A \rangle$  and  $\langle A \rangle$  are not equivalent. To capture the equivalence between effect rows, we therefore model them as multisets.

Additionally, we can extend effect rows with other effect rows. In the syntax of the language, this is specified with the `|` at the end of the effect row: `<A,B|e>` means that the effect row contains `A`, `B` and some (possibly empty) set of remaining effects.

We can use extensions to ensure equivalence between effect rows without specifying the full rows (which might depend on context). For example, the following function uses the `Abort` effect if the called function returns false, while retaining the effects of the wrapped function.

```
1 | let abort_on_false = fn(f: fn() <|e> Bool) <Abort|e> () {
2 |   if f() { () } else { abort() }
3 | }
```

Effect rows need special treatment in the unification algorithm of the type checker, which is detailed in Section 4.1.1.

### 3.2.2 Effect Handlers

To define the implementation of an effect, one has to create a handler for said effect. Handlers are first-class values in Elaine and can be created with the `handler` keyword. They can then be applied to an expression with the `handle` keyword. When `handle` expressions are nested with handlers for the same effect, the innermost `handle` applies.

For example, if we want to use an effect to provide an implicit value, we can make an effect `Val` and a corresponding handler, which `resumes` execution with some values. The `resume` function represents the continuation of the program after the operation. Since handlers are first-class values, we can return the handler from a function to simplify the code. This pattern is quite common to create dynamic handlers with small variations.

```
1 | use std;
2 |
3 | effect Val {
4 |   val() Int
5 | }
6 |
7 | let hVal = fn(x) {
8 |   handler {
9 |     return(x) { x }
10 |    val() { resume(x) }
11 |   }
12 | };
13 |
14 | let main = {
15 |   let a = handle[hVal(6)] add(val(), val());
16 |   let b = handle[hVal(10)] add(val(), val());
17 |   add(a, b)
18 | };
```

The handlers we have introduced for `Val` all call the `resume` function, but that is not required. Conceptually, all effect operations are executed by the `handle`, hence, if we return from the operation, we return from the `handle`. A handler therefore has great control over control flow.

The `Abort` effect uses this mechanism. It defines a single operation `abort`, which returns from the handler without resuming. To show the flexibility that the framework of algebraic effect handlers, provide we will demonstrate several possible handlers for `Abort`. The first ignores the result of the computation, but still halts execution.

```
1 effect Abort {
2   abort() a
3 }
4
5 let hAbort = handler {
6   return(x) { () }
7   abort() { () }
8 };
9
10 let main = {
11   handle[hAbort] {
12     abort();
13     f()
14   };
15   g()
16 };
```

In the program above, `f` will not get called because `hAbort` does not call the continuation, but `g` will be called, because it is used outside the **handle** construct.

Alternatively, we can define a handler that defines a default value for failing expressions. In this example, the handler acts much like an exception handler.

```
1 let hAbort = fn(default) {
2   handler {
3     return(x) { x }
4     abort() { default }
5   }
6 };
7
8 let safe_div = fn(x, y) <Abort> Int {
9   if eq(y, 0) {
10     abort()
11   } else {
12     div(x, y)
13   }
14 };
15
16 let main = add(
17   handle[hAbort(0)] safe_div(3, 0),
18   handle[hAbort(0)] safe_div(10, 2),
19 );
```

We can also map the `Abort` effect to the `Maybe` monad, which is the canonical implementation.

**TODO:** Even for small handlers I need custom data types

```
1 let hAbort = handler Abort {
2   return(x) { Just(x) }
3   abort() { Nothing() }
4 };
```

Finally, we can ignore `abort` calls if we are writing an application in which we always want to try to continue execution no matter what errors occur.<sup>2</sup>

---

<sup>2</sup>With a `never` type, an alternative definition of `Abort` is possible where this handler is not permitted by

```

1 | let hAbort = handler Abort {
2 |   return(x) { x }
3 |   abort() { resume() }
4 | };

```

Just like we can ignore the continuation, we can also call it multiple times, which is useful for non-determinism and logic programming. Listing 3.1 contains the full code for a (very naive) SAT solver in Elaine. We first define a `Yield` effect, so we can yield multiple values from the computation. We will use this to find all possible combinations of boolean inputs that satisfy our equation. The `Logic` effect has two operations. The `branch` operation will call the continuation twice; once with `false` and once `true`. With `fail`, we can indicate that a branch has failed. To find all solutions, we just `branch` on all inputs and `yield` when a correct solution has been found and `fail` when the equation is not satisfied. In listing 3.1, we check for solutions of the equation  $\neg a \wedge b$ .

### 3.3 Higher-Order Effects in Elaine

Higher-order effects in Elaine are supported via elaborations, as proposed by [bach\\_poulsen\\_hefty\\_2023](#). To distinguish higher-order effects from algebraic effects, we write them with a `!` suffix. This syntax was chosen to be reminiscent of macros in Rust, since elaborations are syntactic substitutions and hence behave much like macros. However, it should be noted that they do not behave exactly the same.

Just like algebraic effects have the `handler` and `handle` keywords, to create and apply handlers, higher-order effects have the `elaboration` and `elab` keyword.

Elaborations do not get access to the `resume` function, because they always resume exactly once. Consider the effect `Val` and its elaborated counterpart `Val!`, which calls the continuation with a constant value. With a handler, we have to call `resume`, but we do not need to do that in an elaboration.

<pre> 1   <b>handler</b> { 2     <b>val</b>() { <b>resume</b>(5) } 3   } </pre>	<pre> 1   <b>elaboration</b> Val! -&gt; &lt;&gt; { 2     <b>val</b>() { 5 } 3   } </pre>
---	--

Of course, the goal of elaborations is not to write handlers differently, but to encode higher-order effects. Since elaborations elaborate into algebraic effects, a row of algebraic effects must be specified. As a silly example, we can elaborate the higher-order `Val!` into `Val`, simply replacing occurrences of `val!()` with `val()`;

```

1 | effect Val! {
2 |   val!(): Int
3 | }
4 |
5 | effect Val {
6 |   val(): Int
7 | }
8 |
9 | let hVal = fn(x) {
10 |   handler {
11 |     val() { resume(x) }
12 |   }
13 | };

```

the type system. The signature of `abort` would then be `abort() !`, where `!` is the never type and then `resume` could not be called.

```
1 use std;
2
3 effect Yield {
4   yield(String) ()
5 }
6
7 effect Logic {
8   branch() Bool
9   fail() a
10 }
11
12 let hYield = handler {
13   return(x) { "" }
14   yield(m) { concat(concat(m, "\n"), resume()) }
15 };
16
17 let hLogic = handler {
18   return(x) { () }
19   branch() {
20     resume(true);
21     resume(false)
22   }
23   fail() { () }
24 };
25
26 let show_bools = fn(a, b, c) {
27   let a = concat(show_bool(a), ", ");
28   let b = concat(show_bool(b), ", ");
29   concat(concat(a, b), show_bool(c))
30 };
31
32 let f = fn(a, b, c) { and(not(a), b) };
33
34 let assert = fn(f, a, b, c) <Logic,Yield> () {
35   if f(a, b, c) {
36     yield(show_bools(a, b, c))
37   } else {
38     fail()
39   }
40 };
41
42 let main = handle[hYield] handle[hLogic] {
43   assert(f, branch(), branch(), branch());
44 };
```

Listing 3.1: A naive SAT solver in Elaine.



```

14
15 let eVal = elaboration Val! -> <Val> {
16   val!() { val() }
17 };
18
19 let main = handle[hVal] elab[eVal] val!();

```

The higher-order operations differ from other functions and algebraic operations because they have call-by-name semantics; the arguments are not evaluated before they are passed to the elaboration. Hence, the arguments can be computations, even effectful computations.

This allows us to manipulate computations directly. For example, it is possible to wrap the computation in a handler within an elaboration.

```

1 use std;
2
3 effect Ask {
4   ask(): Int
5 }
6
7 effect Local! {
8   local!(fn(Int) Int, c) : c
9 }
10
11 let eLocal = elaboration Local! -> <Ask> {
12   local!(f, c) {
13     handle[hAsk(f(ask()))] c
14   }
15 };
16
17 let hAsk = fn(x) {
18   handler {
19     ask() { resume(5) }
20   }
21 };
22
23 let main = handle[hAsk(5)] elab[eLocal] local!(
24   fn(x) { mul(2, x) },
25   add(ask(), ask())
26 );

```

This is how higher-order operations such as `local` and `catch` are supported in Elaine.



## Chapter 4

# Elaine Specification

**TODO:** This needs to become more of a story. Maybe it should be integrated with the previous chapter?

This chapter contains the detailed specification for Elaine: the syntax, semantics, the type inference rules and finally some specifics on the type checker that deviate from standard Hindley-Milner type checking.

### 4.1 Syntax definition

The Elaine syntax was designed to be relatively easy to parse. The grammar is white-space insensitive and most constructs are unambiguously identified with keywords at the start.

Based on the previous chapters, the `elab` without an elaboration might be surprising. The use of that syntax is explained in Chapter 5.

The full syntax definition is given in Figure 4.1. For convenience, we define and use several extensions to BNF:

- tokens are written in `monospace` font, this includes the tokens `[]`, `<>`, `|` and `!`, which might be confused with the syntax of BNF,
- `[p]` indicates that the sort `p` is optional,
- `p...p` indicates that the sort `p` can be repeated zero or more times, and
- `p, ..., p` indicates that the sort `p` can be repeated zero or more times, separated by commas.

#### 4.1.1 Effect row semantics

Before explaining the typing judgments of Elaine, let us examine effect rows. The effect row of a computation type determines the context in which the computation can be evaluated. For example, a computation with effect row `<A,B,C>` is valid in a function with effect row `<A,B,C>`. Additionally, the effect rows `<A,B>` and `<B,A>` should be considered to be equivalent.

One possible treatment is then to model effect rows as sets. However, as noted by [leijen\\_koka\\_2014](#), this leads to some problems. Consider the following (abridged) program.

```
1 let v: fn(f: fn() <abort|e> a) e a {  
2   handle[hAbort] f()  
3 };  
4  
5 let main = handle[hAbort] v(fn() { abort() });
```

$$\begin{aligned}
&\text{program } p ::= d \dots d \\
&\text{declaration } d ::= [\text{pub}] \text{ mod } x \{d \dots d\} \\
&\quad | [\text{pub}] \text{ use } x; \\
&\quad | [\text{pub}] \text{ let } p = e; \\
&\quad | [\text{pub}] \text{ effect } \phi \{s, \dots, s\} \\
&\quad | [\text{pub}] \text{ type } x \{s, \dots, s\} \\
&\text{block } b ::= \{ es \} \\
&\text{expression list } es ::= e; es \\
&\quad | \text{let } p = e; es \\
&\quad | e \\
&\text{expression } e ::= x \\
&\quad | () \mid \text{true} \mid \text{false} \mid \text{number} \mid \text{string} \\
&\quad | \text{fn}(p, \dots, p) [T] b \\
&\quad | \text{if } e \text{ b else } b \\
&\quad | e(e, \dots, e) \mid \phi(e, \dots, e) \\
&\quad | \text{handler } \{\text{return}(x) b, o, \dots, o\} \\
&\quad | \text{handle}[e] e \\
&\quad | \text{elaboration } x! \rightarrow \Delta \{o, \dots, o\} \\
&\quad | \text{elab}[e] e \mid \text{elab } e \\
&\quad | es \\
&\text{annotatable variable } p ::= x : T \mid x \\
&\text{signature } s ::= x(T, \dots, T) T \\
&\text{effect clause } o ::= x(x, \dots, x) b \\
&\text{type } T ::= \Delta \tau \mid \tau \\
&\text{value type } \tau ::= x \\
&\quad | () \mid \text{Bool} \mid \text{Int} \mid \text{String} \\
&\quad | \text{fn}(T, \dots, T) T \\
&\quad | \text{handler } x \tau \tau \\
&\quad | \text{elaboration } x! \Delta \\
&\text{effect row } \Delta ::= \langle \phi, \dots, \phi[|x] \rangle \\
&\text{effect } \phi ::= x \mid x!
\end{aligned}$$

Figure 4.1: Syntax definition of Elaine

The function  $v$  “removes” an **abort** effect from the effect row. By treating the effect row as a set, there would be no **abort** effect in return type of  $v$ . However, in **main**, there is another handler for **abort** and hence **abort** should be in the effect row.

The treatment of effect rows then simplifies if duplicated effects are allowed (leijen\_koka\_2014). Hence we use multisets to model effect rows, meaning that the row  $\langle A, B, B, C \rangle$  is represented by the multiset  $\{A, B, B, C\}$ . This yields a semantics where the multiplicity of effects is significant, but the order is not.

Since the effect row of a computation must match the effect row of the context in which it is used, the effect row of the computation is an overapproximation of the effects that are necessary. Therefore, we should allow effect row polymorphism, so that the same expression can be used within multiple contexts.

Effect row polymorphism is enabled via the *row tail*, which is denoted with the  $|$  symbol followed by an identifier.

The  $|$  symbol signifies extension of the effect row with another (possibly arbitrary) effect row. We determine compatibility between effect rows by unifying them. That is

We define the operation set as follows:

$$\begin{aligned} \text{set}(\varepsilon) &= \text{set}(\langle \rangle) = \emptyset \\ \text{set}(\langle A_1, \dots, A_n \rangle) &= \{A_1, \dots, A_n\} \\ \text{set}(\langle A_1, \dots, A_n | R \rangle) &= \text{set}(\langle A_1, \dots, A_n \rangle) + \text{set}(R). \end{aligned}$$

Note that the extension uses the sum, not the union of the two sets. This means that  $\text{set}(\langle A | \langle A \rangle \rangle)$  should yield  $\{A, A\}$  instead of  $\{A\}$ .

Then we get the following equality relation between effect rows  $A$  and  $B$ :

$$A \cong B \iff \text{set}(A) = \text{set}(B).$$

In typing judgments, the effect row is an overapproximation of the effects that actually used by the expression. We freely use set operations in the typing judgments, implicitly calling the the set function on the operands where required. An omitted effect row is treated as an empty effect row  $\langle \rangle$ .

Any effect prefixed with a  $!$  is a higher-order effect, which must be elaborated instead of handled. Due to this distinction, we define the operations  $H(R)$  and  $A(R)$  representing the higher-order and first-order subsets of the effect rows, respectively. The same operators are applied as predicates on individual effects, so the operations on rows are defined as:

$$H(\Delta) = \{\phi \in \Delta \mid H(\phi)\} \quad \text{and} \quad A(\Delta) = \{\phi \in \Delta \mid A(\phi)\}.$$

**TODO:** Talk about (leijen\_extensible\_2005; leijen\_koka\_2014).

During type checking effect rows are represented as a pair consisting of a multiset of effects and an optional extension variable. In this section we will use a more explicit notation than the syntax of Elaine by using the multiset representation directly. Hence, a row  $\langle A_1, \dots, A_n | e_A \rangle$  is represented as the multiset  $\{A_1, \dots, A_n\} + e_A$ .

Like with regular Hindley-Milner type inference, two rows can be unified if we can find a substitution of effect row variables that make the rows equal. For effect rows, this yields 3 distinct cases.

If both rows are closed (i.e. have no extension variable) there are no variables to be substituted and we just employ multiset equality. That is, to unify rows  $A$  and  $B$  we check that  $A = B$ . If that is true, we do not need to unify further and unification has succeeded. Otherwise, we cannot make any substitutions to make them equal and unification has failed.

If one of the rows is open, then the set of effects in that row need to be a subset of the effects in the other row. To unify the rows

$$A + e_A \quad \text{and} \quad B$$

we assert that  $A \subseteq B$ . If that is true, we can substitute  $e_n$  for the effects in  $B - A$ .

Finally, there is the case where both rows are open:

$$A + e_A \quad \text{and} \quad B + e_B.$$

In this case, unification is always possible, because both rows can be extended with the effects of the other. We create a fresh effect row variable  $e_C$  with the following substitutions:

$$\begin{aligned} e_A &\rightarrow (B - A) + e_C \\ e_B &\rightarrow (A - B) + e_C. \end{aligned}$$

In other words,  $A$  is extended with the effects that are in  $B$  but not in  $A$  and similarly,  $B$  is extended with the effects in  $A$  but not in  $A$ .

## 4.2 Typing judgments

The context  $\Gamma = (\Gamma_M, \Gamma_V, \Gamma_E, \Gamma_\Phi)$  consists of the following parts:

$\Gamma_M : x \rightarrow (\Gamma_V, \Gamma_E, \Gamma_\Phi)$	module to context
$\Gamma_V : x \rightarrow \sigma$	variable to type scheme
$\Gamma_E : x \rightarrow (\Delta, \{f_1, \dots, f_n\})$	higher-order effect to elaboration type
$\Gamma_\Phi : x \rightarrow \{s_1, \dots, s_n\}$	effect to operation signatures

**INFO:** A  $\Gamma_T$  for data types might be added.

Whenever one of these is extended, the others are implicitly passed on too, but when declared separately, they not implicitly passed. For example,  $\Gamma''$  is empty except for the single  $x : T$ , whereas  $\Gamma'$  implicitly contains  $\Gamma_M, \Gamma_E$  &  $\Gamma_\Phi$ .

$$\Gamma'_V = \Gamma_V, x : T \quad \Gamma''_V = x : T$$

If the following invariants are violated there should be a type error:

- The operations of all effects in scope must be disjoint.
- Module names are unique in every scope.
- Effect names are unique in every scope.

### 4.2.1 Type inference

We have the usual generalize and instantiate rules. But, the generalize rule requires an empty effect row.

**QUESTION:** Koka requires an empty effect row. Why?

$$\frac{\Gamma \vdash e : \sigma \quad \alpha \notin \text{ftv}(\Gamma)}{\Gamma \vdash e : \forall \alpha. \sigma} \quad \frac{\Gamma \vdash e : \forall \alpha. \sigma}{\Gamma \vdash e : \sigma[\alpha \mapsto T']}$$

Where  $\text{ftv}$  refers to the free type variables in the context.

### 4.2.2 Expressions

We freely write  $\tau$  to mean that a type has an empty effect row. That is, we use  $\tau$  and a shorthand for  $\langle \rangle \tau$ . The  $\Delta$  stands for an arbitrary effect row. We start with everything but the handlers and elaborations and put them in a separate section.

$$\frac{\Gamma_V(x) = \Delta \tau}{\Gamma \vdash x : \Delta \tau} \quad \frac{\Gamma \vdash e : \Delta \tau}{\Gamma \vdash \{e\} : \Delta \tau} \quad \frac{\Gamma \vdash e_1 : \Delta \tau \quad \Gamma_V, x : \tau \vdash e_2 : \Delta \tau'}{\Gamma \vdash \text{let } x = e_1; e_2 : \Delta \tau'}$$

$$\overline{\Gamma \vdash () : \Delta ()} \quad \overline{\Gamma \vdash \text{true} : \Delta \text{Bool}} \quad \overline{\Gamma \vdash \text{false} : \Delta \text{Bool}}$$

$$\frac{\Gamma_V, x_1 : T_1, \dots, x_n : T_n \vdash c : T \quad T_i = \langle \rangle \tau_i}{\Gamma \vdash \text{fn}(x_1 : T_1, \dots, x_n : T_n) T \{e\} : \Delta (T_1, \dots, T_n) \rightarrow T}$$

$$\frac{\Gamma \vdash e_1 : \Delta \text{Bool} \quad \Gamma \vdash e_2 : \Delta \tau \quad \Gamma \vdash e_3 : \Delta \tau}{\Gamma \vdash \text{if } e_1 \{e_2\} \text{ else } \{e_3\} : \Delta \tau}$$

$$\frac{\Gamma \vdash e : (\tau_1, \dots, \tau_n) \rightarrow \Delta \tau \quad \Gamma \vdash e_i : \Delta \tau_i}{\Gamma \vdash e(e_1, \dots, e_n) : \Delta \tau}$$

### 4.2.3 Declarations and Modules

The modules are gathered into  $\Gamma_M$  and the variables that are in scope are gathered in  $\Gamma_V$ . Each module has a the type of its public declarations. Note that these are not accumulative; they only contain the bindings generated by that declaration. Each declaration has the type of both private and public bindings. Without modifier, the public declarations are empty, but with the `pub` keyword, the private bindings are copied into the public declarations.

$$\frac{\Gamma_{i-1} \vdash m_i : \Gamma_{m_i} \quad \Gamma_{M,i} = \Gamma_{M,i-1}, \Gamma_{m_i}}{\Gamma_0 \vdash m_1 \dots m_n : ()}$$

$$\frac{\Gamma_{i-1} \vdash d_i : (\Gamma'_i; \Gamma'_{\text{pub},i}) \quad \Gamma_i = \Gamma_{i-1}, \Gamma'_i \quad \Gamma \vdash \Gamma'_{\text{pub},1}, \dots, \Gamma'_{\text{pub},n}}{\Gamma_0 \vdash \text{mod } x \{d_1 \dots d_n\} : (x : \Gamma)}$$

$$\frac{\Gamma \vdash d : \Gamma'}{\Gamma \vdash d : (\Gamma'; \varepsilon)} \quad \frac{\Gamma \vdash d : \Gamma'}{\Gamma \vdash \text{pub } d : (\Gamma'; \Gamma')} \quad \overline{\Gamma \vdash \text{import } x : \Gamma_M(x)}$$

$$\frac{f_i = \forall \alpha. (\tau_{i,1}, \dots, \tau_{i,n_i}) \rightarrow \alpha x \quad \Gamma'_V = x_1 : f_1, \dots, x_m : f_m}{\Gamma \vdash \text{type } x \{x_1(\tau_{1,1}, \dots, \tau_{1,n_1}), \dots, x_m(\tau_{m,1}, \dots, \tau_{m,n_m})\} : \Gamma'}$$

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash \text{let } x = e : (x : T)}$$

#### 4.2.4 First-Order Effects and Handlers

Effects are declared with the **effect** keyword. The signatures of the operations are stored in  $\Gamma_\Phi$ . The types of the arguments and resumption must all have no effects.

A handler must have operations of the same signatures as one of the effects in the context. The names must match up, as well as the number of arguments and the return type of the expression, given the types of the arguments and the resumption. The handler type then includes the handled effect  $\phi$ , an “input” type  $\tau$  and an “output” type  $\tau'$ . In most cases, these will be at least partially generic.

The handle expression will simply add the handled effect to the effect row of the inner expression and use the the input and output type.

$$\frac{s_i = op_i(\tau_{i,1}, \dots, \tau_{i,n_i}) : \tau_i \quad \Gamma'_\Phi(x) = \{s_1, \dots, s_n\}}{\Gamma \vdash \mathbf{effect} \ x \ \{s_1, \dots, s_n\} : \Gamma'}$$

$$\frac{\Gamma \vdash e_h : \mathbf{handler} \ \phi \ \tau \ \tau' \quad \Gamma \vdash e_c : \langle \phi | \Delta \rangle \ \tau}{\Gamma \vdash \mathbf{handle} \ e_h \ e_c : \Delta \ \tau'}$$

$$\frac{\begin{array}{l} A(\phi) \quad \Gamma_\Phi(\phi) = \{s_1, \dots, s_n\} \quad \Gamma, x : \tau \vdash e_{\text{ret}} : \tau' \\ \left[ \begin{array}{l} s_i = x_i(\tau_{i,1}, \dots, \tau_{i,m_i}) \rightarrow \tau_i \quad o_i = x_i(x_{i,1}, \dots, x_{i,m_i}) \{e_i\} \\ \Gamma_V, \text{resume} : (\tau_i) \rightarrow \tau', x_{i,1} : \tau_{i,1}, \dots, x_{i,i_m} : \tau_{i,i_m} \vdash e_i : \tau' \end{array} \right]_{1 \leq i \leq n} \end{array}}{\Gamma \vdash \mathbf{handler} \ \{\text{return}(x)\{e_{\text{ret}}\}, o_1, \dots, o_n\} : \mathbf{handler} \ \phi \ \tau \ \tau'}$$

#### 4.2.5 Higher-Order Effects and Elaborations

The declaration of higher-order effects is similar to first-order effects, but with exclamation marks after the effect name and all operations. This will help distinguish them from first-order effects.

Elaborations are of course similar to handlers, but we explicitly state the higher-order effect  $x!$  they elaborate and which first-order effects  $\Delta$  they elaborate into. The operations do not get a continuation, so the type checking is a bit different there. As arguments they take the effectless types they specified along with the effect row  $\Delta$ . Elaborations are not added to the value context, but to a special elaboration context mapping the effect identifier to the row of effects to elaborate into.

The **elab** expression then checks that a elaboration for all higher-order effects in the inner expression are in scope and that all effects they elaborate into are handled.

$$\frac{s_i = op_i!(\tau_{i,1}, \dots, \tau_{i,n_i}) : \tau_i \quad \Gamma'_\Phi(x!) = \{s_1, \dots, s_n\}}{\Gamma \vdash \mathbf{effect} \ x! \ \{s_1, \dots, s_n\} : \Gamma'}$$

$$\frac{\begin{array}{l} \Gamma_\Phi(x!) = \{s_1, \dots, s_n\} \quad \Gamma'_E(x!) = \Delta \\ \left[ \begin{array}{l} s_i = x_i!(\tau_{i,1}, \dots, \tau_{i,m_i}) \ \tau_i \quad o_i = x_i!(x_{i,1}, \dots, x_{i,m_i}) \{e_i\} \\ \Gamma, x_{i,1} : \Delta \ \tau_{i,1}, \dots, x_{i,n_i} : \Delta \ \tau_{i,n_i} \vdash e_i : \Delta \ \tau_i \end{array} \right]_{1 \leq i \leq n} \end{array}}{\Gamma \vdash \mathbf{elaboration} \ x! \rightarrow \Delta \ \{o_1, \dots, o_n\} : \Gamma'}$$

**INFO:** Later, we could add more precise syntax for which effects need to be present in the arguments of the elaboration operations.

**INFO:** It is not possible to elaborate only some of the higher-order effects. We could change the behaviour to allow this later.



$$\frac{[\Gamma_E(\phi) \subseteq \Delta]_{\phi \in H(\Delta')} \quad \Gamma \vdash e : \Delta' \tau \quad \Delta = A(\Delta')}{\Gamma \vdash \text{elab } e : \Delta \tau}$$

### 4.3 Desugaring

Fold over the syntax tree with the following operation:

$$\begin{aligned} D(\text{fn}(x_1 : T_1, \dots, x_n : T_n) T \{e\}) &= \lambda x_1, \dots, x_n. e \\ D(\text{let } x = e_1; e_2) &= (\lambda x. e_2)(e_1) \\ D(e_1; e_2) &= (\lambda \_ . e_2)(e_1) \\ D(\{e\}) &= e \\ D(e) &= e \end{aligned}$$

### 4.4 Elaboration resolution

### 4.5 Semantics

#### 4.5.1 Reduction contexts

$$\begin{aligned} E ::= & [] \mid E(e_1, \dots, e_n) \mid v(v_1, \dots, v_n, E, e_1, \dots, e_m) \\ & \mid \text{if } E \{e\} \text{ else } \{e\} \\ & \mid \text{let } x = E; e \mid E; e \\ & \mid \text{handle}[E] e \mid \text{handle}[v] E \\ & \mid \text{elab}[E] e \mid \text{elab}[v] E \end{aligned}$$

$$\begin{aligned} X_{op} ::= & [] \mid X_{op}(e_1, \dots, e_n) \mid v(v_1, \dots, v_n, X_{op}, e_1, \dots, e_m) \\ & \mid \text{if } X_{op} \{e_1\} \text{ else } \{e_2\} \\ & \mid \text{let } x = X_{op}; e \mid X_{op}; e \\ & \mid \text{handle}[X_{op}] e \mid \text{handle}[h] X_{op} \text{ if } op \notin h \\ & \mid \text{elab}[X_{op}] e \mid \text{elab}[\epsilon] X_{op} \text{ if } op! \notin e \end{aligned}$$

#### 4.5.2 Reduction rules

$$\begin{aligned} c(v_1, \dots, v_n) &\longrightarrow \delta(c, v_1, \dots, v_n) \\ &\quad \text{if } \delta(c, v_1, \dots, v_n) \text{ defined} \\ (\lambda x_1, \dots, x_n. e)(v_1, \dots, v_n) &\longrightarrow e[x_1 \mapsto v_1, \dots, x_n \mapsto v_n] \\ \text{if true } \{e_1\} \text{ else } \{e_2\} &\longrightarrow e_1 \\ \text{if false } \{e_1\} \text{ else } \{e_2\} &\longrightarrow e_2 \\ \text{handle}[h] v &\longrightarrow e[x \mapsto v] \\ &\quad \text{where } \text{return}(x)\{e\} \in H \\ \text{handle}[h] X_{op}[op(v_1, \dots, v_n)] &\longrightarrow e[x_1 \mapsto v_1, \dots, x_n \mapsto v_n, \text{resume} \mapsto k] \end{aligned}$$

$$\begin{array}{lcl}
& \text{where } op(x_1, \dots, x_n)\{e\} \in h \\
& \quad k = \lambda y . \text{handle}[h] X_{op}[y] \\
\text{elab}[\epsilon] v & \longrightarrow & v \\
\text{elab}[\epsilon] X_{op!}[op!(e_1, \dots, e_n)] & \longrightarrow & \text{elab}[\epsilon] X_{op!}[e[x_1 \mapsto e_1, \dots, x_n \mapsto e_n]] \\
& \quad \text{where } op!(x_1, \dots, x_n)\{e\} \in \epsilon
\end{array}$$

## Chapter 5

# Implicit Elaboration Resolution

With Elaine, we aim to explore the further ergonomic improvements we can make for programming with effects. We noticed in our experimentation that elaborations are often not parametrized and that there is often only one in scope at a time. Hence, the elaboration for a higher-order effect that should be applied is often non-ambiguous and can be inferred.

This led us to add implicit elaboration resolution to Elaine. This feature allows the programmer to omit the elaboration from the **elab** keyword. Take the following example, where we let the interpreter infer the elaboration.

```
1 effect Val! {  
2   val!() Int  
3 }  
4  
5 let eVal = elaboration Val! -> <> {  
6   val!() { 5 }  
7 };  
8  
9 let main = elab { val!() };
```

This feature allows for some nice programming patterns. For example, it allows us to bundle a higher-order effect and a standard elaboration in a module. When this module is imported, the effect and elaboration are both brought into scope and **elab** will apply the standard elaboration automatically.

The order in which elaborations are applied does not influence the semantics of the program.<sup>[citation needed](#)</sup> Therefore, implicit elaboration resolution can also be used to elaborate multiple effects with a single **elab** construct. To make the inference predictable, we require that an implicit elaboration must elaborate all higher-order effects.

This is a nice convenience, but it requires some caution. A problem arises when we define multiple elaborations for an effect; which one should then be used? We have chosen to simply give a type error in this case. To fix the type error we simply provide the elaboration we want with **elab**[eVal1].

```
1 effect Val! {  
2   val!() Int  
3 }  
4  
5 let eVal1 = elaboration Val! -> <> {  
6   val!() { 1 }  
7 };  
8  
9 let eVal2 = elaboration Val! -> <> {
```

```

10 |   val!() { 2 }
11 | };
12 |
13 | let main = elab { val!() }; # Type error here!

```

The elaboration resolution consists of two parts: inference and transformation. The inference is done by the type checker and is hence type-directed, which records the inferred elaboration. After type checking the program is then transformed such that all implicit elaborations have been replaced by explicit elaborations.

To infer the elaborations, the type checker first analyses the sub-expression. This will yield some computation type with an effect row containing both higher-order and algebraic effects:  $\langle H!_1, \dots, H!_n, A_1, \dots, A_m \rangle$ . It then checks the type environment to look for elaborations  $E_1, \dots, E_n$  which elaborate  $H!_1, \dots, H!_n$ , respectively. Only elaborations that are directly in scope are considered, so if an elaboration resides in another module, it needs be imported first. For each higher-order effect, there must be exactly one elaboration.

The **elab** is finally transformed into one explicit **elab** per higher-order effect. Recall that the order of elaborations does not matter for the semantics of the program, meaning that we safely apply them any order.

```

1 | elab[ $E_1$ ] elab[ $E_2$ ] ... elab[ $E_n$ ]

```

A nice property of this feature is that the transformation results in very readable code. Because the elaboration is in scope, there is an identifier for it in scope as well. The transformation then simply inserts this identifier. The **elab** in the first example of this chapter will, for instance, be transformed to **elab**[eVal]. An IDE could then display this transformed **elab** as an inlay hint.

**TODO:** If Jonathan's syntax highlighting and linking is integrated we can talk about that here too.

The same inference could trivially be added for handlers. However, this would yield to unpredictable results, because the semantics of the program depend on the order in which handlers are applied. If we then have an expression with two algebraic effects, how do we determine the order in which they should be applied?

There are some solutions for this. For example, we could require that the sub-expression can only use a single algebraic effect, but that would make the feature much less useful. Another possibility is to assign some standard precedence to effects. We think that this would become quite confusing in the end.

Another difficulty with using inference for handlers is that handlers are often parametrized and that there is then not just a handler in scope, but only a function returning a handler. This makes inference impossible in most cases.

## Chapter 6

# Elaboration Compilation

Since Elaine has novel semantics for elaborations, it is worth examining its relation to well-studied constructs from programming language theory. We therefore defined a transformation from elaborations into handlers, translating higher-order effects into algebraic effects, while preserving their semantics.

The goal of this transformation is twofold. First, it further connects hefty algebras and Elaine to existing literature. For example, by compiling to a representation with only algebraic effects, we can then further compile the program using existing techniques, such as the compilation procedures defined for Koka (leijen\_type\_2017). Second, the transformation allows us to encode elaborations in existing libraries and languages for algebraic effects.

In this thesis and the accompanying implementation, we only provide the first step: transforming elaborations into handlers.

### 6.1 Observations about Elaborations

Examining the semantics of elaborations, we observe that elaborations are similar to macros: they perform a syntactic substitution. For instance, the program on the left transforms into the program on the right by replacing `plus_two!`, with the expression `{ x + 2 }`.

```
1 use std;
2
3 effect PlusTwo! {
4   plus_two!(Int)
5 }
6
7 let ePlusTwo = {
8   elaboration PlusTwo! -> <> {
9     plus_two!(x) { add(x, 2) }
10  }
11 };
12
13 let main = elab plus_two!(5);
```

```
1 let main = { add(5, 2) };
```

Additionally, the location of the `elab` does not matter as long as the operations are contained within it. For instance, these expressions are equivalent:

```
1 let main = elab[e] {
2   a!();
3   a!()
4 };
1 let main = {
2   elab[e] a!();
3   elab[e] a!()
4 };
```

In some cases, it should therefore be possible to match up elaborations and operations at compile-time. However, this does not hold in general. A clear hurdle is that we might have a complex expression for the elaboration, such as an **if** expression:

```
1 | elab[if cond { elab1 } else { elab2 }] c
```

In this example, the elaboration might depend on a run-time value, making compile-time substitution difficult.

Moreover, a single operation might need to be elaborated by different **elab** constructs, depending on run-time computations. In the listing below, there are two elaborations **eOne** and **eTwo** of an effect with the operation **a!()**. The **a!()** operation in **f** is elaborated where **f** is called. If the condition **k** evaluates to **true**, **f** is assigned to **g**, which is elaborated by **eOne**. However, if **k** evaluates to **false**, **f** is called in the inner **elab** and hence **a!()** is elaborated by **eTwo**.

```
1 | elab[eOne] {
2 |   let g = elab[eTwo] {
3 |     let f = fn() { a!() };
4 |     if k {
5 |       f
6 |     } else {
7 |       f();
8 |       fn() { () }
9 |     }
10 |   }
11 |   g()
12 | }
```

Therefore, the analysis of determining the elaboration that should be applied to an operation is non-local. The substitution might still be a nice optimization or simplification step, but it cannot guarantee that the transformed program will not contain higher-order effects.

Instead, the elaborations can be localized with a technique similar to dictionary-passing style. Any function with higher-order effects then takes the elaboration to apply as an argument and the operation is wrapped in an **elab**. The elaboration is then taken from **elabA** at the call-site.

```
1 | let elabA = eOne;
2 | let g = {
3 |   let elabA = eTwo;
4 |   let f = fn(e) { elab[e] a!() };
5 |   if k {
6 |     f
7 |   } else {
8 |     f(elabA);
9 |     fn() { () }
10 |   }
11 | }
12 | g(elabA)
```

## 6.2 Encoding Macros as Functions

**TODO:** Macros are functions with arguments and return value transformed wrapped in functions.

The main difference between a function and a macro-like operation in our semantics is that macro arguments are thunked. This holds because the macro is type-checked independently