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

Version of June 27, 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

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 27, 2023

Contents

Chapter 1

Introduction

As a program runs, it usually interacts with its environment. A program 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 1989b).

Pure functions are the computations which always return identical values for identical inputs and do not interact with their environment. While some programs can be viewed as pure, analysing only pure computation leaves out many aspects of programs, such as side effects, non-determinism and non-termination (Moggi 1989b). So, in practice, many programs are *impure* or *effectful*.

We can extend this analysis to the type checker of a programming language. Such a type checker not only checks the types of values, but also the effects that a computation might have. This helps programmers reason about effects in a program.

Not sure where to put this and how to add more motivation.

While many effects does not seem to have anything in common at first glance, they do share an important property: they yield control to some other procedure. citation needed This procedure, called an *effect handler*, implements the necessary operations to perform the effect. An effect handler could be a memory allocator, a scheduler, a kernel, an exception handler or something else. Swapping out one effect handler for another then changes the semantics of the computation. This modularity of effect handlers is a powerful concept, allowing us to reuse computations within different contexts. For example, we might want to swap out one asynchronous runtime for another, or we could replace writing to `stdout` with writing to `stderr`.

Better examples?

Some programming languages opt to give the programmer virtually unrestricted access to effectful operations. For instance, any part of a C program can interact with memory, the filesystem or the network. The program can even yield control to any location with the `goto` keyword, which has famously been criticized by Dijkstra (1968). This "anything goes" approach puts a large burden of ensuring correct behaviour of effects on the programmer.

Many programming languages have done so by adding dedicated features for specific effects, such as exceptions, coroutines and generators. These features are useful, but they lack flexibility and modularity: any effect must be backed by the language and new effects cannot be created without adding a new feature to the language. It is also not usually possible to change the behaviour of these features.

A different approach is taken in many languages adhering to the functional programming paradigm: they choose to represent effects with monads. In theory, monads are a perfect representation for effects, since effects originally defined by Moggi (1991) as monads. Yet, mathematical rigour does not make a practical programming language. In practice, monads suffer from a lack of modularity and flexibility. citation needed

This modularity is at the core of the theory of *algebraic effects*, which unifies many common effects into a single concept (Plotkin and Power 2001; Plotkin and Pretnar 2009). In languages with algebraic effects, such as Koka (Leijen 2014), Eff (Bauer and Pretnar

2015), Frank (Lindley, McBride, and McLaughlin 2017), and Effekt (Brachthäuser, Schuster, and Ostermann 2020), effect handlers can be modularly defined within the language. The programmer can freely declare new effects and handlers. An effect here consists of a set of *effect operations*, which yield control to their corresponding handler, which then performs the operation. The handler can resume the computation by calling the *continuation*, which is a function that represents the rest of the computation. 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 (Brachthäuser, Schuster, and Ostermann 2020).

However, there are some commonly used effects that are not algebraic, namely *higher-order effects*. That is, effect operations that take effectful computations as arguments that do not behave as continuations. Several approaches to modelling higher-order effects have been proposed (van den Berg et al. 2021; Wu, Schrijvers, and Hinze 2014). Most recently, Bach Poulsen and van der Rest (2023) have proposed to extend algebraic effects with *hefty algebras*, which introduces *effect elaborations* to define higher-order effects.

In this thesis, we introduce a novel programming language called *Elaine*. The core idea of Elaine is to define a language which features elaborations and higher-order effects as a first-class construct. This brings the theory of hefty algebras into practice. With Elaine, we aim to demonstrate the usefulness of elaborations as a language feature. Throughout this thesis, we present example programs with higher-order effects to argue that elaborations are a natural and easy representation of higher-order effects.

Like handlers for algebraic effects, elaborations require the programmer to specify which elaboration should be applied. However, elaboration have several properties which make it likely that there is only one relevant possible elaboration. Hence, we argue that elaboration instead should often be implicit and inferred by the language. To this end, we introduce *implicit elaboration resolution*, a novel feature that infers an elaboration from the variables in scope.

Finally, we give two transformations from higher-order effects to algebraic effects. There are two reasons for defining such a transformation. The first is to show how elaborations can be compiled in a larger compilation pipeline. The second is that these transformations show how elaborations could be added to existing systems for algebraic effects.

We present a specification for Elaine, including the syntax definition, typing judgments and semantics. Along with this specification, we provide a reference implementation written in Haskell in the artefact accompanying this thesis. This implementation includes a parser, type checker, interpreter, pretty printer, and the transformations mentioned above.

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 (Section 4.1).
- We provide a set of examples for programming with higher-order effect operations.
- We present a type system for a language with higher-order effects and elaborations, based on Hindley-Milner type inference and inspired by the Koka type system. This type system introduces 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 procedure for this inference (Chapter 5).
- We define two transformations form programs with elaborations and higher-order effects to programs with only handlers and algebraic effects. The first transformation is convenient, but relies on impredicativity and therefore only works in languages that allow impredicativity, such as Elaine, Haskell and Koka. The second transformation is more involved, but does not rely on impredicativity and would therefore also be allowed in a language like Agda.

1.2 Artefact

TODO: Describe contents and structure of artefact

The artefact is available online at `https://github.com/tertsdiepraam/thesis/elaine`.

Chapter 2

Preliminaries

FEEDBACK: This chapter should be less about the fundamentals and more about the things that are relevant to the rest of the thesis.

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 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: λ-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 λ-calculus, effects are not modelled; every function is a function in the mathematical sense, that is, a pure computation (Moggi 1989b). Hence, many observable properties of programs are ignored, such as non-determinism and side effects. In their seminal paper, Moggi (1989b) unified *monads* with computational effects, which they initially called notions of computation. Moggi identified that for any monad $T : C \to C$ 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 and Wadler (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.

FEEDBACK: The meaning of this is unclear

FEEDBACK: "technique" is mysterious

FEEDBACK: Monad transformers should be moved to related work and in a separate section.

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 1989a). 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.

Well it is, but not a "combined" monad. Not sure how to describe that. I guess that the monad `A (B a)` is not a monad over `a`, but only over `B a`. Need to explain why that is a problem

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.

FEEDBACK: translate handling to the context of monad transformers

FEEDBACK: Should be

2.2 Algebraic Effects

TODO: Add a section on effect rows

FEEDBACK: The notation should be more consistent.

Algebraic effects have emerged as another model for effects. In this model, effects are treated as *algebraic theories*.

2.2.1 Algebraic Theories

This section introduces algebraic theories. In particular, we will discuss algebraic theories with parametrized operations and general arities. This forms a foundation on which we can define algebraic effects. For a more complete introduction to algebraic theories, we refer to Bauer (2018).

TODO: Replace formal definitions with definitions more in line with the rest of the thesis.

Definition 1 (Algebraic theory)**.** An algebraic theory (or *equational theory*) is a pair $\mathsf{T} = (\Sigma_{\mathsf{T}}, \mathcal{E}_{\mathsf{T}})$ consisting of a signature Σ_{T} and a collection $\mathcal{E}_{\mathsf{T}}$ of Σ_{T}-equations. The signature Σ_{T} is a consist of a set of operations and their corresponding arities. The T-equations define what laws should hold for these operations.

For example, we can define the suggestively-named operations `mul` with arity 2, `inv` with arity 1 and `zero` with arity 0. We can then build a collection of equations that specify the behaviour of these operations. If we want them to behave like a group, we need the laws of associativity, identity and inverse:

$$\begin{aligned}
\mathsf{add}(x, \mathsf{add}(y, z)) &= \mathsf{add}(\mathsf{add}(x, y), z)\\
\mathsf{add}(x, \mathsf{zero}) &= x\\
\mathsf{add}(x, \mathsf{inv}(x)) &= \mathsf{zero}\\
\mathsf{add}(\mathsf{inv}(x), x) &= \mathsf{zero}
\end{aligned}$$

Hence, the operations, arities and these equations form the algebraic theory for a group.

An algebraic theory is 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. That is, they need to be given associated functions that define their implementation.

Definition 2 (Model)**.** A *model* of an algebraic theory T is an interpretation of Σ_{T} for which all the equations in $\mathcal{E}_T$ hold.

Crucially, we can create a *free model* for any algebraic theory for which a valid model exists. This free model is constructed by interpreting the operations as a tree where the arguments to an operation are its children in the tree. In programming language terminology, we are interpreting the operations as an abstract syntax tree.

2.2.2 Notation for Computations

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

$$op(p_1, \ldots, p_n, \lambda x.\kappa)$$

for an effectful operation op, with parameters $p_1, \ldots, p_n$ and some continuation κ. This notation follows the continuation passing style of programming, where the operation is evaluated first and then calls the continuation with some value.

However, denoting the continuation as an argument of the operations does not match how we tend to think of operations. For a more intuitive notation, we introduce a sequencing operator:

FEEDBACK: Why not do-notation or bind operator? Answer: Make it similar to elaine. But I could change it to do-notation or bind.

$$x \leftarrow op(p_1, \ldots, p_n);\ \kappa.$$

When the value passed to the continuation is discarded, we will omit the variable assignment and write

$$op(p_1, \ldots, p_n); \kappa.$$

This gives us a notation that is close to what an effectful program might look like, but is still rooted in the representation from algebraic theories.

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

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

then first performs the `put` and then the `get` operation, returning the result of the `get` operation. This can be equivalently written as

$$\texttt{put}(a, \lambda\_.\texttt{get}()),$$

which is more in line with the notation for operations from algebraic theories.

2.2.3 Effects as Algebraic Theories

Plotkin and Power (2001) have shown that many effects can be represented as algebraic theories. To do so, we have to define a signature and a set of equations for a given effect.

We start with the signature for `State`:

FEEDBACK: What are signatures? The arrow is undefined?

$$\texttt{put} : S \to \mathbf{1} \quad \text{and} \quad \texttt{get} : \mathbf{1} \to S.$$

This signature indicates that `put` takes some value of type S as parameter and resumes with the unit () and that `get` takes () and calls the continuation with a value of type S. Now we can define the equations that we want `State` to follow:

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

This gives us an algebraic theory corresponding to the `State` monad. Plotkin and Power (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, iteration, cooperative asynchronicity, traversal, input and output citation needed. These effects are called *algebraic effects*.

FEEDBACK: State monad undefined (could be part of the monad section)

FEEDBACK: We need to define the State/Reader/Exception monads.

As shown by Plotkin and Power (2003), an effect is algebraic if and only if it satisfies the *algebraicity property*, which can be expressed as follows when $m_1, \ldots, m_n$ are computation parameters:

$$x \leftarrow op(m_1, \ldots, m_n);\ \kappa \quad = \quad op(\{x \leftarrow m_1;\ \kappa\}, \ldots, \{x \leftarrow m_n;\ \kappa\}).$$

Therefore, sequencing the continuation must distribute over the operations. In other words, all computation parameters must be *continuation-like*, that is, they are some computation followed by the continuation (Bach Poulsen and van der Rest 2023).

FEEDBACK: Emphasize Plotkin & Power instead of Bach Poulsen

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 that

$$\mathtt{local}(f, m) \gg\!= \mathtt{ask}() \quad \neq \quad \mathtt{local}(f, m \gg\!= \mathtt{ask}()),$$

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:

$$\mathtt{catch}(m_1, m_2) \gg\!= \mathtt{throw}() \quad \neq \quad \mathtt{catch}(m_1 \gg\!= \mathtt{throw}(), m_2 \gg\!= \mathtt{throw}()).$$

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. The two sides therefore have different semantics and hence the `catch` effect is not algebraic.

2.2.4 Effect Handlers

FEEDBACK: This needs to be expanded.

The distinction between effects which are and which are not algebraic has been described as the difference between *effect constructors* and *effect deconstructors* (Plotkin and Power 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.

Plotkin and Pretnar (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:

$$\begin{aligned} hAsk(v) = \mathtt{handler}\{\, & \mathsf{return}(x) \mapsto x, \\ & \mathtt{ask}()\ \kappa \mapsto \kappa(v)\}. \end{aligned}$$

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

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

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

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

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, Schrijvers, and Hinze 2014).

2.3 Elaborations

Several solutions to the modularity problem have been proposed (van den Berg et al. 2021; Wu, Schrijvers, and Hinze 2014). Most recently, Bach Poulsen and van der Rest (2023) introduced hefty algebras. The idea behind hefty algebras is that there is an additional layer of modularity, 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:

FEEDBACK: This needs explanation

FEEDBACK: which definition?

$$
\begin{aligned}
eLocal \quad &\stackrel{\text{def}}{=} \quad \texttt{elaboration}\ \{ \\
&\qquad \texttt{local!}(f, m) \mapsto \{v \leftarrow \texttt{ask}();\ \texttt{handle}[hAsk(f(v))]\ m\} \\
&\quad \},
\end{aligned}
$$

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

FEEDBACK: language we're working in is unclear

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

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

$$
\begin{aligned}
&\texttt{handle}[hAsk(5)]\ \{ \\
&\quad x \leftarrow \texttt{ask}(); \\
&\quad y \leftarrow \{ \\
&\quad\quad v \leftarrow \texttt{ask}(); \\
&\quad\quad \texttt{handle}[hAsk((\lambda x.\ 2 \cdot x)(v))]\ \texttt{ask}() \\
&\quad \}; \\
&\quad x + y \\
&\}
\end{aligned}
$$

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

FEEDBACK: give example

TODO: Add some stuff about typing for effects, i.e. effect rows.

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. The basic feature of elaborations has been extended with two novel features: implicit elaboration resolution and compilation of elaborations, which are explained in Chapters 5 and 6, respectively.

This chapter introduces Elaine with motivating examples for the design choices. The full specification is given in Chapter 4. More example programs are available in the artifact accompanying this thesis.

FEEDBACK: Overview of what can be found in the artifact.

3.1 Basics

The design of Elaine is similar to Koka, with syntactical elements inspired by Rust. Apart from the elaborations and handlers, it features let bindings, modules, 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.

Listing 3.1 contains a program that uses the basic features of Elaine and prints whether the square of 4 is even or odd.

3.2 Data Types

TODO: Write this section. Short version: we can define data types and match on the variants.

3.3 Algebraic Effects

The programs in the previous section are all pure and contain no effects. Like the languages discussed in Chapter 7, Elaine additionally has first class support for effects and effect han-

Listing 3.1: A simple Elaine program. The result of this program is the string "The square of 4 is even".

```
# The standard library contains basic functions for manipulation
# of integers, booleans and strings.
use std;

# Functions are created with `fn` and bound with `let`, just like
# other values. The last expression in a function is returned.
let square = fn(x: Int) Int {
   mul(x, x)
};

let is_even = fn(x: Int) Bool {
   eq(0, modulo(x, 2))
};

# Type annotations can be inferred:
let square_is_even = fn(x) {
   let result = is_even(square(x));
   if result { "even" } else { "odd" }
};

let give_answer = fn(f, s, x) {
      let prefix = concat(concat(s, " "), show_int(x));
   let text = concat(prefix, " is ");
   let answer = f(x);
   concat(text, answer)
};

let main = give_answer(square_is_even, "The square of", 4);
```

dlers.

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 3.3.2).

3.3.1 Effect Rows

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 `<>`. For example, an integer has type `<> Int`. Without row elision, the `square` function in the previous section could therefore have been written as

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

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 `<A,B> Int`.

Important here is that this type is equivalent to `<B,A> Int`: the order of effects in effect rows is irrelevant. However, the multiplicity is important, that is, the effect rows `<A,A>` and `<A>` 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.

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

Effect rows need special treatment in the unification algorithm of the type checker, which is detailed in Section 4.2.

3.3.2 Effect Handlers

To define the implementation of an effect, we have to define a handler it. 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 `resume`s 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.

```
use std;

effect Val {
    val() Int
}

let hVal = fn(v) {
    handler {
        return(x) { x }
        val() { resume(v) }
    }
};

let main = {
    let a = handle[hVal(6)] add(val(), val());
    let b = handle[hVal(10)] add(val(), val());
    add(a, b)
};
```

Calling the `resume` function is not required. All effect operations are executed by the **`handle`** expression, hence, if we return from the operation, we return from the **`handle`** expression.

The `Abort` effect is an example which does not call the continuation. It defines a single operation `abort`, which stops the evaluation of the computation. To show the modularity that the framework of algebraic effect handlers, provide we will demonstrate several possible handlers for `Abort`. First, we have the canonical handler for `Abort`, which returns the `Maybe` monad. If the computation returns, it returns the returned value wrapped in `Just`. If the computation aborts, it returns `Nothing()`.

```
effect Abort {
   abort() ()
}

let hAbort = handler {
   return(x) { Just(x) }
   abort() { Nothing() }
};

let main = handle[hAbort] {
   abort();
   5
};
```

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.

```
let hAbort = fn(default) {
   handler {
      return(x) { x }
      abort() { default }
   }
};

let safe_div = fn(x, y) <Abort> Int {
   if eq(y, 0) {
      abort()
   } else {
      div(x, y)
   }
};

let main = add(
   handle[hAbort(0)] safe_div(3, 0),
   handle[hAbort(0)] safe_div(10, 2),
);
```

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.[1]

```
let hAbort = handler {
```

[1] With a never type, an alternative definition of `Abort` is possible where this handler is not permitted by the type system. The signature of `abort` would then be `abort() !`, where `!` is the never type and then `resume` could not be called.

```
    return(x) { x }
    abort() { resume(()) }
};
```

Just like we can ignore the continuation, we can also call it multiple times, which is useful for non-determinism and logic programming. Listing 3.2 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 the formula. 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 formula is not satisfied. In listing 3.2, we check for solutions of the equation $\neg a \wedge b$.

3.4 Higher-Order Effects

TODO: This section needs to be expanded a lot.

TODO: Figure out why LaTeXwants to put all the listings at the end of the chapter.

Higher-order effects in Elaine are supported via elaborations, as proposed by Bach Poulsen and van der Rest (2023). To distinguish higher-order effects and operations from algebraic effects and operations, we write them with a `!` suffix. 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.

Just like we have the **`handler`** and **`handle`** keywords to create and apply handlers for algebraic effects, we can create and apply elaborations with the **`elaboration`** and **`elab`** keywords. Unlike handlers, elaborations do not get access to the `resume` function, because they always resume exactly once.

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

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

Listing 3.2: A naive SAT solver in Elaine using algebraic effects to branch the execution.

```
use std;

effect Yield {
  yield(String) ()
}

effect Logic {
   branch() Bool
   fail() a
}

let hYield = handler {
   return(x) { "" }
   yield(m) {
      concat(concat(m, "\n"), resume(()))
   }
};

let hLogic = handler {
   return(x) { () }
   branch() {
      resume(true);
      resume(false)
   }
   fail() { () }
};

let show_bools = fn(a, b, c) {
   let a = concat(show_bool(a), ", ");
   let b = concat(show_bool(b), ", ");
   concat(concat(a, b), show_bool(c))
};

let f = fn(a, b, c) { and(not(a), b) };

let assert = fn(f, a, b, c) <Logic,Yield> () {
   if f(a, b, c) {
      yield(show_bools(a, b, c))
   } else {
      fail()
   }
};

let main = handle[hYield] handle[hLogic] {
   assert(f, branch(), branch(), branch());
};
```

Listing 3.3: Reader effect with higher-order `local` operation in Elaine.

```
use std;

effect Ask {
   ask() Int
}

effect Reader! {
   local!(fn(Int) Int, a) a
}

let hAsk = fn(v: Int) {
   handler {
      return(x) { x }
      ask() { resume(v) }
   }
};

let eLocal = elaboration Reader! -> <Ask> {
   local!(f, c) {
      handle[hAsk(f(ask()))] c
   }
};

let double = fn(x) { mul(2, x) };

let main = handle[hAsk(2)] elab[eLocal] {
   local!(double, add(ask(), ask()));
};
```

Chapter 4

Elaine Specification

TODO: Custom type declarations are in the language but not explained in this chapter yet.

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,
- $p \dots p$ indicates that the sort p can be repeated zero or more times, and
- p`,` $\dots$ `,` p indicates that the sort p can be repeated zero or more times, separated by commas.

4.2 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 (2014), this leads to some problems. Consider the following (abridged) program.

```
let v: fn(f: fn() <abort|e> a) e a {
   handle[hAbort] f()
};

let main = handle[hAbort] v(fn() { abort() });
```

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

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

TODO: Talk about (Leijen 2005, 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:

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

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.

4.3 Typing judgments

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

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

INFO: A Γ_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, Γ'' is empty except for the single $x : T$, whereas Γ' implicitly contains Γ_M, Γ_E & Γ_Φ.

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

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.

4.3.1 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?

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

Where ftv refers to the free type variables in the context.

4.3.2 Expressions

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

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

$$\frac{}{\Gamma \vdash \texttt{()} : \Delta\ \texttt{()}} \qquad \frac{}{\Gamma \vdash \texttt{true} : \Delta\ \texttt{Bool}} \qquad \frac{}{\Gamma \vdash \texttt{false} : \Delta\ \texttt{Bool}}$$

$$\frac{\Gamma_V, x_1 : T_1, \ldots, x_n : T_n \vdash c : T \qquad T_i = \langle\rangle\tau_i}{\Gamma \vdash \texttt{fn}(x_1 : T_1, \ldots, x_n : T_n)\ T\ \{e\} : \Delta\ (T_1, \ldots, T_n) \to T}$$

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

$$\frac{\Gamma \vdash e : (\tau_1, \ldots, \tau_n) \to \Delta\ \tau \qquad \Gamma \vdash e_i : \Delta\ \tau_i}{\Gamma \vdash e(e_1, \ldots, e_n) : \Delta\ \tau}$$

4.3.3 Declarations and Modules

The modules are gathered into Γ_M and the variables that are in scope are gathered in Γ_V. Each module has 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.

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

$$\frac{\Gamma_{i-1} \vdash d_i : (\Gamma_i'; \Gamma_{\mathrm{pub},i}') \qquad \Gamma_i = \Gamma_{i-1}, \Gamma_i' \qquad \Gamma \vdash \Gamma_{\mathrm{pub},1}', \ldots, \Gamma_{\mathrm{pub},n}'}{\Gamma_0 \vdash \texttt{mod}\ x\ \{d_1 \ldots d_n\} : (x : \Gamma)}$$

$$\frac{\Gamma \vdash d : \Gamma'}{\Gamma \vdash d : (\Gamma'; \varepsilon)} \qquad \frac{\Gamma \vdash d : \Gamma'}{\Gamma \vdash \texttt{pub}\ d : (\Gamma'; \Gamma')} \qquad \frac{}{\Gamma \vdash \texttt{import}\ x : \Gamma_M(x)}$$

$$\frac{\begin{array}{c} f_i = \forall\alpha.(\tau_{i,1}, \ldots, \tau_{i,n_i}) \to \alpha\ x \\ \Gamma_V' = x_1 : f_1, \ldots, x_m : f_m \end{array}}{\Gamma \vdash \texttt{type}\ x\ \{x_1(\tau_{1,1}, \ldots, \tau_{1,n_1}), \ldots, x_m(\tau_{m,1}, \ldots, \tau_{m,n_m})\} : \Gamma'}$$

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

4.3.4 Algebraic Effects and Handlers

Effects are declared with the `effect` keyword. The signatures of the operations are stored in Γ_Φ. 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 ϕ, an "input" type τ and an "output" type τ'. 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 input and output type.

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

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

$$\frac{\begin{matrix} A(\phi) \qquad \Gamma_\Phi(\phi) = \{s_1, \dots, s_n\} \qquad \Gamma, x : \tau \vdash e_{\text{ret}} : \tau' \\ \begin{bmatrix} s_i = x_i(\tau_{i,1}, \dots, \tau_{i,m_i}) \to \tau_i \qquad o_i = x_i(x_{i,1}, \dots, x_{i,m_i})\ \{e_i\} \\ \Gamma_V, \mathit{resume} : (\tau_i) \to \tau', x_{i,1} : \tau_{i,1}, \dots, x_{i,i_m} : \tau_{i,i_m} \vdash e_i : \tau' \end{bmatrix}_{1 \le i \le n} \end{matrix}}{\Gamma \vdash \texttt{handler}\ \{\texttt{return}(x)\{e_{\text{ret}}\}, o_1, \dots, o_n\} : \texttt{handler}\ \phi\ \tau\ \tau'}$$

4.3.5 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 Δ 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 Δ. 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.

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 higher-order effects. We could change the behaviour to allow this later.

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.

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

$$\frac{\begin{matrix} \Gamma_\Phi(x!) = \{s_1, \dots, s_n\} \qquad \Gamma'_E(x!) = \Delta \\ \begin{bmatrix} s_i = x_i!(\tau_{i,1}, \dots, \tau_{i,m_i})\ \tau_i \qquad o_i = x_i!(x_{i,1}, \dots, x_{i,m_i})\{e_i\} \\ \Gamma, x_{i,1} : \Delta\ \tau_{i,1}, \dots, x_{i,n_i} : \Delta\ \tau_{i,n_i} \vdash e_i : \Delta\ \tau_i \end{bmatrix}_{1 \le i \le n} \end{matrix}}{\Gamma \vdash \texttt{elaboration}\ x! \to \Delta\ \{o_1, \dots, o_n\} : \Gamma'}$$

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

4.4 Desugaring

To simplify the reduction rules, we simplify the AST by desugaring some constructs. This transform is given by a fold over the syntax tree with the following operation:

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

4.5 Semantics

The semantics of Elaine are defined as reduction semantics.

We use two separate contexts to evaluate expressions. The E context is for all constructs except effect operations, such as **if**, **let** and function applications. The X_{op} context is the context in which a handler can reduce an operation *op*.

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

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

TODO: Add some explanation

$$\begin{aligned}
c(v_1, \dots, v_n) &\longrightarrow \delta(c, v_1, \dots, v_n) \\
&\qquad\quad \text{if}\ \delta(c, v_1, \dots, v_n)\ \text{defined} \\
(\lambda x_1, \dots, x_n.e)(v_1, \dots, v_n) &\longrightarrow e[x_1 \mapsto v_1, \dots, x_n \mapsto v_n] \\
\texttt{if true}\ \{e_1\}\ \texttt{else}\ \{e_2\} &\longrightarrow e_1 \\
\texttt{if false}\ \{e_1\}\ \texttt{else}\ \{e_2\} &\longrightarrow e_2
\end{aligned}$$

$$\texttt{handle}[h]\ v \longrightarrow e[x \mapsto v]$$

$$
\begin{array}{rcl}
 & & \quad \text{where } \mathtt{return}(x)\{e\} \in H \\
\mathtt{handle}[h]\ X_{op}[op(v_1, \ldots, v_n)] & \longrightarrow & e[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n, \mathit{resume} \mapsto k] \\
 & & \quad \text{where } op(x_1, \ldots, x_n)\{e\} \in h \\
 & & \qquad k = \lambda y\ .\ \mathtt{handle}[h]\ X_{op}[y] \\
\mathtt{elab}[\epsilon]\ v & \longrightarrow & v \\
\mathtt{elab}[\epsilon]\ X_{op!}[op!(e_1, \ldots, e_n)] & \longrightarrow & \mathtt{elab}[\epsilon]\ X_{op!}[e[x_1 \mapsto e_1, \ldots, x_n \mapsto e_n]] \\
 & & \quad \text{where } op!(x_1, \ldots, x_n)\{e\} \in \epsilon
\end{array}
$$

4.6 Standard Library

Elaine does not include any operators. This choice was made to simplify parsing of the language. For the lack of operators, any manipulation of primitives needs to be done via the standard library of built-in functions.

These functions reside in the `std` module, which can be imported like any other module with the **`use`** statement to bring its contents into scope.

The full list of functions available in the `std` module, along with their signatures and descriptions, is given in Figure 4.2.

	Name	Type signature		Description
Arithmetic	`add`	**`fn`**`(Int, Int)`	`Int`	addition
	`sub`	**`fn`**`(Int, Int)`	`Int`	subtraction
	`neg`	**`fn`**`(Int)`	`Int`	negation
	`mul`	**`fn`**`(Int, Int)`	`Int`	multiplication
	`div`	**`fn`**`(Int, Int)`	`Int`	division
	`modulo`	**`fn`**`(Int, Int)`	`Int`	modulo
	`pow`	**`fn`**`(Int, Int)`	`Int`	exponentiation
Comparisons	`eq`	**`fn`**`(Int, Int)`	`Bool`	equality
	`neq`	**`fn`**`(Int, Int)`	`Bool`	inequality
	`gt`	**`fn`**`(Int, Int)`	`Bool`	greater than
	`geq`	**`fn`**`(Int, Int)`	`Bool`	greater than or equal
	`lt`	**`fn`**`(Int, Int)`	`Bool`	less than
	`leq`	**`fn`**`(Int, Int)`	`Bool`	less than or equal
Boolean operations	`not`	**`fn`**`(Bool)`	`Bool`	boolean negation
	`and`	**`fn`**`(Bool, Bool)`	`Bool`	boolean and
	`or`	**`fn`**`(Bool, Bool)`	`Bool`	boolean or
String operations	`concat`	**`fn`**`(Bool, Bool)`	`Bool`	string concatenation
Conversions	`show_int`	**`fn`**`(Int)`	`String`	integer to string
	`show_bool`	**`fn`**`(Bool)`	`String`	integer to string

Figure 4.2: Overview of the functions in the `std` module in Elaine.

Chapter 5

Implicit Elaboration Resolution

FEEDBACK: The process is not interesting for readers. Stick to the definition and motivation.

With Elaine, we aim to explore the further ergonomic improvements we can make for programming with effects. We note that elaborations are often not parametrized and that there is often only one in scope at a time. Hence, when we encounter an `elab`, there is only one possible elaboration that could be applied.

Therefore, we propose that, in this situation, the language should allow the elaboration to be inferred. Take the example in Chapter 5, where we let Elaine infer the elaboration for us.

```
let eLocal = elaboration Reader! -> <Ask> {
   local!(f, c) {
      handle[hAsk(f(ask()))] c
   }
};

let main = handle[hAsk(2)] elab {
   local!(double, ask());
};
```

A use case of this feature is when an effect and elaboration are defined in the same module. When this module is imported, the effect and elaboration are both brought into scope and **`elab`** will apply the standard elaboration automatically.

```
mod local {
   pub effect Ask { ... }
   pub let hAsk = handler { ... }
   pub effect Reader! { ... }
   pub let eLocal = elaboration Reader! -> <Ask> { ... }
}

use local;

# We don't have to specify the elaboration, since it is
# imported along with the effect.
let main = handle[hAsk] elab { local!(double, ask!()) };
```

The order in which elaborations are applied does not influence the semantics of the program. Therefore, implicit elaboration resolution can also be used to elaborate multiple effects

This needs justification.

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 multiple elaborations for an effect are in scope; which one should then be used? To keep the result of the inference predictable and deterministic, we give a type error in this case. Hence, we know that, if type checking succeeds, the inference procedure has found exactly one elaboration to apply for each higher-order effect. If not, we simply write the elaboration explicitly.

```
let eLocal1 = elaboration Local! -> <> { ... };
let eLocal2 = elaboration Local! -> <> { ... };

let main = elab { local!(double, ask!()) }; # 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.

FEEDBACK: bad sentence

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: $\langle H!_1, \ldots, H!_n, A_1, \ldots, A_m \rangle$. It then checks the type environment to look for elaborations $E_1, \ldots, E_n$ which elaborate $H!_1, \ldots, H!_n$, 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.

```
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 a novel semantics for elaborations, it is worth examining its relation to well-studied constructs from programming language theory. Therefore, we introduce a transformation from programs with higher-order effects to a program with only algebraic effects, 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 2017). In this thesis and the accompanying implementation, we provide the first step of this compilation. Second, the transformation allows us to encode elaborations in existing libraries and languages for algebraic effects.

6.1 Non-locality of Elaborations

TODO: Actually it's not just non-locality but also undecidable

Examining the semantics of elaborations, we observe that elaborations 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 }`.

```
use std;

effect PlusTwo! {
   plus_two!(Int)
}

let ePlusTwo = {
   elaboration PlusTwo! -> <> {
      plus_two!(x) { add(x, 2) }
   }
};

let main = elab plus_two!(5);
```

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

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

```
let main = elab[e] {
   a!();
   a!()
};
```

```
let main = {
   elab[e] a!();
   elab[e] a!()
};
```

In some cases, it is therefore possible to statically determine the elaboration that should be applied. In that situation, we can remove the elaboration from the program by performing the syntactic substitution.

However, we cannot apply that technique in general. One example where it does not work is when the elaboration is given by a complex expression, such as an **`if`**-expression:

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

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 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`.

```
elab[eOne] {
   let g = elab[eTwo] {
      let f = fn() { a!() };
      if k {
         f
      } else {
         f();
         fn() { () }
      }
   }
   g()
}
```

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

6.2 Operations as Functions

As explained in Section 4.5, higher-order operations are evaluated differently from functions. The main difference is that the arguments are thunked and passed by name, instead of by value.

This behaviour can be emulated for functions if anonymous functions are passed as arguments instead of expressions. That is, for any operation call `op!(e1, ..., eN)`, we wrap the arguments into functions to get `op(fn() { e1 }, ..., fn() { eN })`. In the body of `op`, we then replace each occurrence of an argument `x` with `x()` such that the thunked value is obtained. The `op` operation can then be evaluated like a function instead, but it still has the intended semantics.

6.3 Compiling Elaborations to Dictionary Passing

Instead, the elaborations can be transforms with a technique similar to dictionary-passing style: the implicit context of elaborations is explicitly passed to functions that require a higher-order effect.

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.

Listing 6.1: Some explanation. Should be higher-order?

```
use std;

effect A! {
   arithmetic!(Int, Int) Int
}

let eAdd = elaboration A! -> <> {
   arithmetic!(a, b) { add(a, b) }
};

let eMul = elaboration A! -> <> {
   arithmetic!(a, b) { mul(a, b) }
};

# The original program
let foo = fn(k) {
   elab[eAdd] {
      let g = elab[eMul] {
         let f = fn() { a!(5, 2 + 3) };
         if k {
            f
         } else {
            let x = f();
            fn() { x }
         }
      };
      g()
   }
};

# Create a new type with one constructor to represent the
# elaboration. The fields of the constructor are the
# operations. We assume for convenience that all the
# generated identifiers do not conflict with existing
# identifiers.
type ElabA {
   ElabA(fn(fn() Int, fn() Int) Int)
}

# Convenience function to access the operation a from A!
let elab_A_a = fn(e: ElabA) {
   let ElabA(v) = e;
   v
};

let eAdd = ElabA ( fn(a, b) { add(a(), b()) } );
let eMul = ElabA ( fn(a, b) { mul(a(), b()) } );

# The transformed program
let foo' = fn(k) {
   let elab_A = eAdd;
   let g = {
      let elab_A = eMul;
      let f = fn(elab_A) { elab_A_a(elab_a)(fn() { 5 }, fn() { 2 + 3) } };
      if k {
```

6.4 Handlers as Dictionary Passing

We now have two pieces of the puzzle: we know how to translate the semantics for higher-order operations into regular functions, and we have determined that we need something like dictionary passing to determine what elaboration should be applied. The problem is that our language – as currently defined – does not support dictionaries. Even if dictionaries were defined, the type safety of the transformed program would be difficult to check without a sub-structural type system.

This leads to the final observation: effect handlers can be used as our dictionary passing construct. Conceptually, both **`elab`** and **`handle`** work similarly: they define a scope in which a given elaboration or handler is used. This scope is the same for both. We start by defining a handler that returns an elaboration for higher-order effect `A!`, much like the `Ask` effect from Chapter 3.

```
let hElab = fn(e) {
   handler {
      return(x) { x }
      askElabA() { resume(e) }
   }
};

handle[hElab(eOne)] {
   let g = handle[hElab(eTwo)] {
      let f = fn(e) { elab[askElabA()] a!() };
      if k {
         f
      } else {
         f();
         fn() { () }
      }
   }
   g()
}
```

TODO: Define the transformation.

6.5 Compiling Elaborations into Handlers

Combining the ideas above, we obtain a surprisingly simple transformation. Each elaboration is transformed into a handler, which resumes with a function containing the original expression, where argument occurrences force the thunked values. Since elaborations are now handlers, we need to change the **`elab`** constructs to **`handle`** constructs accordingly. Finally, the arguments to operation calls are thunked and the function that is resumed is called, that is, there is an additional `()` at the end of the operation call.

```
elab[e_1] {e_2}   ⟹   handle[e_1] {e_2}
```

```
elaboration {                                handler {
    op_1!(x_{1,1}, ..., x_{k_1,1}) { e_1 }       op_1(x_{1,1}, ..., x_{k_1,1}) {
    ...                                   ⟹         resume(fn() {e_1[x_{i,1} ↦ x_{i,1}()]})
    op_n!(x_{1,n}, ..., x_{k_n,n}) { e_n }       }
}                                                ...
                                                 op_n(x_{1,n}, ..., x_{k_n,n}) {
                                                     resume(fn() {e_1[x_{i,n} ↦ x_{i,n}()]})
                                                 }
                                             }
```

```
op_j!(e_1, ..., e_k)   ⟹   op_j(fn() \{e_1\}, ..., fn() \{e_k\})()
```

6.6 An Alternative Design for Elaine

The simplicity of the transformation makes it alluring and begs the question: are dedicated language features for higher-order effects necessary or is a simpler approach possible?

Since we can encode elaborations as handlers, we can write higher-order effects as the result of the transformation above directly. Below is an example of an exception effect written in this style.

```
effect Exc {
   catch(fn() <Throw> a, fn() a)
}

let hExc = handler {
   return(x) { x }
   catch(f, g) {
      resume(fn() {
         let res = handle[hThrow] f();
         match res {
            Just(x) => x,
            Nothing => g(),
         }
      })
   }
};

let main = handle[hCatch] catch(
   fn() { ... },
   fn() { ... },
)();
```

Now let us introduce a few (hypothetical) syntactic conveniences. First, like Koka, we let `{...}`| represent `fn(){...}`. Second, we allow the `return` case to be omitted and default to the identity function. Third, if an operation ends with `!`, the operation body is wrapped in `resume({...})` and the operations is given an underscore prefix with the original name brought into scope defined as

```
let op! = fn(x_1, ..., x_n){ _op!(x_1, ..., x_n)() };
```

Our previous example then reduces to

```
effect Exc {
   catch!(fn() <Throw> a, fn() a)
}

let hExc = handler {
   catch!(f, g) {
      let res = handle[hThrow] f();
      match res {
         Just(x) => x,
         Nothing => g(),
      }
   }
};

let main = handle[hExc] catch!(
   { ... },
   { ... },
);
```

The result is almost as convenient as the version of Elaine presented in the previous chapters and does not have the split between elaborations and handlers, which possibly makes it easier to understand. The same design could be applied to other implementations of algebraic effects as well with relative ease.

To illustrate that point, below is modular `catch` effect in Koka. While the handler is certainly more verbose than handlers for algebraic effects in Koka, the implementation is also quite simple. Koka could implement a shorthand for creating the boilerplate and essentially the same functionality as Elaine.

```
effect abort
  ctl abort(): a

effect exc
  fun catch_( f : () -> <abort|e> a, g : () -> e a ) : (() -> e a)

val hAbort = handler
  return(x) Just(x)
  ctl abort() Nothing

val hExc = handler
  fun catch_(f, g)
    fn()
      match hAbort(f)
        Just(x) -> x
        Nothing -> g()

fun catch(f, g)
  catch_(f, g)()

fun main()
  with hExc
  println(catch({ if True then abort() else 5 }, { 0 }))
```

Because we can then “emulate” higher-order effects, the importance for explicit support for them is reduced. An advantage of this approach is that, because we do not have to elaborate into algebraic effects, we do not have to handle them. In Elaine the `main` function looks like this:

```
let main = handle[hAbort] elab catch!(if true { abort() } else { 5 }, { 0 });
```

TODO: Talk about impredicativity and how that makes it so that it does not work.

Chapter 7

Related Work

FEEDBACK: Can be expanded. Provide context for this thesis. What have others done, what's missing, and what does this thesis add?

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 2005).

TODO: Monad transformers

As the theoretical research around effects has progressed, new libraries and languages have emerged using the state-of-the-art effect theories. These frameworks can be divided into two categories: effects encoded in existing type systems and effects as first-class features.

These implementations provide ways to define, use and handle effectful operations. Additionally, many implementations provide type level information about effects via *effect rows*. These are extensible lists of effects that are equivalent up to reordering. The rows might contain variables, which allows for *effect row polymorphism*.

7.0.1 Effects as Monads

There are many examples of libraries like this for Haskell, including `fused-effects`[1], `polysemy`[2], `freer-simple`[3] and `eff`[4]. Each of these libraries give the encoding of effects a slightly different spin in an effort to find the most ergonomic and performant representation.

As explained in Chapter 2, monads correspond with effectful computations. Any language in which monads can be expressed therefore has some support for effects. Languages that encourage a functional style of programming have embraced this framework in particular.

Haskell currently features an `IO` monad (Peyton Jones and Wadler 1993) as well as a large collection of monads and monad transformers available via libraries, such as `mtl`[5]. This is notable, because there is a strong connection between monad transformers and algebraic effects (Schrijvers et al. 2019).

Algebraic effects have also been encoded in Haskell, Agda and other languages. The key to this encoding is the observation that the sum of two algebraic theories yields an algebraic theory. This theory then again corresponds to a monad. In particular, we can construct a `Free` monad to model the theory (Kammar, Lindley, and Oury 2013; Swierstra 2008).

We can therefore define a polymorphic `Free` monad as follows:

```
data Free f a
  = Pure a
  | Do (f (Free f a))
```

[1] `https://github.com/fused-effects/fused-effects`
[2] `https://github.com/polysemy-research/polysemy`
[3] `https://github.com/lexi-lambda/freer-simple`
[4] `https://github.com/hasura/eff`
[5] `https://github.com/haskell/mtl`

The parameter `f` here can be a sum of effect operations, which forms the effect row. This yields some effect row polymorphism, but the effect row cannot usually be reordered. To compensate for this lack of reordering, many libraries define typeclass constraints that can be used to reason about effects in effect rows.

Effect rows are often constructed using the *Data Types à la Carte* technique (Swierstra 2008), which requires a fairly robust typeclass system. Hence, many languages cannot encode effects within the language itself. In some languages, it is possible to work around the limitations with metaprogramming, such as the Rust library `effin-mad`[6], though the result does not integrate well with the rest of language and its use is discouraged by the author.

Using the `eff` Haskell library as an example, we get the following function signature for an effectful function that accesses the filesystem:

```
readfile :: FileSystem :< effs => String -> Eff effs String
```

In this signature, `FileSystem` is an effect and `effs` is a polymorphic tail. The signature has a constraint stating that the `FileSystem` effect should be in the effect row `effs`. This means that the `readfile` function must be called in a context at least wrapped in a handler for the `FileSystem` effect.

Contrast the signature above with a more conventional signature of `readfile` using the `IO` monad:

```
readfile :: String -> IO String
```

FEEDBACK: But it's too coarse grained!

This signature is more concise and arguably easier to read. Therefore, while libraries for algebraic effects offer semantic improvements over monads (and monad transformers), they are limited in the syntactic sugar they can provide.

FEEDBACK: Feedback to self: this is a non-sequitur

However, the ergonomics of these libraries depend on the capabilities of the type system of the language. Since the effects are encoded as a monad, a monadic style of programming is still required. For both versions of `readline`, we can use the function the same way. For example, a function that reads the first line from a file might be written as below.

```
firstline filename = do
  res <- readfile filename
  return $ head $ lines $ res
```

Some of these libraries support *scoped effects* (Wu, Schrijvers, and Hinze 2014), which is a limited but practical frameworks for higher-order effects. It can express the `local` and `catch` operations, but some higher-order effects are not supported.

Add λ-abstraction as example.

7.0.2 First-class Effects

The motivation of add effects to a programming language is twofold. First, we want to explore how to integrate effects into languages with type systems in which effects cannot be natively encoded. Second, built-in effects allow for more ergonomic and performant implementations. Naturally, the ergonomics of any given implementation are subjective, but we undeniably have more control over the syntax by adding effects to the language. For example, a language might include the previously mentioned implicit `do`-notation

Notable examples of languages with support for algebraic effects are Eff (Bauer and Pretnar 2015), Koka (Leijen 2014), Idris (Brady 2013) and Frank (Lindley, McBride, and McLaughlin 2017), which are alWe can write the `readfile` signature and `firstline` function from before in Koka as follows:

QUESTION: any simple examples?

```
fun readfile( s : string ) : <filesystem> string

```

[6] `https://github.com/rosefromthedead/effing-mad`

```
fun firstline( s: string ) : <filesystem> maybe<string>
   head(lines(readfile(s)))
```

From this example, we can see that the syntactic overhead of the effect rows is much smaller than what is provided by the Haskell libraries. Furthermore, the monadic style of programming is no longer necessary in Koka.

Effect row variables can be used to ensure the same effects across multiple functions without specifying what they are. This is especially useful for higher-order functions. For example, we can ensure that `map` has the same effect row as its argument:

```
fun map ( xs : list<a>, f : a -> e b ) : e list<b>
   ...
```

Other languages choose a more implicit syntax for effect polymorphism. Frank (Lindley, McBride, and McLaughlin 2017) opts to have the empty effect row represent the *ambient effects*. The signature of `map` is then written as

```
map : {X -> []Y} -> List X -> []List Y
```

Since Koka's representation is slightly more explicit, we will be using that style throughout this paper. Elaine's row semantics are inspired by Koka's and are explained in Chapter 3.

Several extensions to algebraic effects have been explored in the languages mentioned above. Koka supports scoped effects and named handlers (Xie et al. 2022), which provides a mechanism to distinguish between multiple occurrences of an effect in an effect row.

Chapter 8

Conclusion

TODO:

Bibliography

Bach Poulsen, Casper and Cas van der Rest (Jan. 9, 2023). "Hefty Algebras: Modular Elaboration of Higher-Order Algebraic Effects". In: *Proceedings of the ACM on Programming Languages* 7 (POPL), pp. 1801–1831. ISSN: 2475-1421. DOI: 10.1145/3571255. URL: https://dl.acm.org/doi/10.1145/3571255 (visited on 01/26/2023).

Bauer, Andrej (2018). "What is algebraic about algebraic effects and handlers?" In: Publisher: arXiv Version Number: 2. DOI: 10.48550/ARXIV.1807.05923. URL: https://arxiv.org/abs/1807.05923 (visited on 06/04/2023).

Bauer, Andrej and Matija Pretnar (Jan. 2015). "Programming with algebraic effects and handlers". In: *Journal of Logical and Algebraic Methods in Programming* 84.1, pp. 108–123. ISSN: 23522208. DOI: 10.1016/j.jlamp.2014.02.001. URL: https://linkinghub.elsevier.com/retrieve/pii/S2352220814000194 (visited on 04/03/2023).

Van den Berg, Birthe et al. (2021). "Latent Effects for Reusable Language Components". In: *Programming Languages and Systems*. Ed. by Hakjoo Oh. Vol. 13008. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, pp. 182–201. ISBN: 978-3-030-89050-6 978-3-030-89051-3. DOI: 10.1007/978-3-030-89051-3_11. URL: https://link.springer.com/10.1007/978-3-030-89051-3_11 (visited on 01/12/2023).

Brachthäuser, Jonathan Immanuel, Philipp Schuster, and Klaus Ostermann (Nov. 13, 2020). "Effects as capabilities: effect handlers and lightweight effect polymorphism". In: *Proceedings of the ACM on Programming Languages* 4 (OOPSLA), pp. 1–30. ISSN: 2475-1421. DOI: 10.1145/3428194. URL: https://dl.acm.org/doi/10.1145/3428194 (visited on 06/02/2023).

Brady, Edwin (Sept. 25, 2013). "Programming and reasoning with algebraic effects and dependent types". In: *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*. ICFP'13: ACM SIGPLAN International Conference on Functional Programming. Boston Massachusetts USA: ACM, pp. 133–144. ISBN: 978-1-4503-2326-0. DOI: 10.1145/2500365.2500581. URL: https://dl.acm.org/doi/10.1145/2500365.2500581 (visited on 06/13/2023).

Dijkstra, Edsger W. (Mar. 1968). "Letters to the editor: go to statement considered harmful". In: *Communications of the ACM* 11.3, pp. 147–148. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/362929.362947. URL: https://dl.acm.org/doi/10.1145/362929.362947 (visited on 05/31/2023).

Kammar, Ohad, Sam Lindley, and Nicolas Oury (Sept. 25, 2013). "Handlers in action". In: *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*. ICFP'13: ACM SIGPLAN International Conference on Functional Programming. Boston Massachusetts USA: ACM, pp. 145–158. ISBN: 978-1-4503-2326-0. DOI: 10.1145/2500365.2500590. URL: https://dl.acm.org/doi/10.1145/2500365.2500590 (visited on 01/18/2023).

Leijen, Daan (July 23, 2005). "Extensible records with scoped labels". In.

Leijen, Daan (June 5, 2014). “Koka: Programming with Row Polymorphic Effect Types”. In: *Electronic Proceedings in Theoretical Computer Science* 153, pp. 100–126. ISSN: 2075-2180. DOI: 10.4204/EPTCS.153.8. URL: http://arxiv.org/abs/1406.2061v1 (visited on 06/16/2023).

— (Jan. 2017). “Type directed compilation of row-typed algebraic effects”. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL ’17: The 44th Annual ACM SIGPLAN Symposium on Principles of Programming Languages. Paris France: ACM, pp. 486–499. ISBN: 978-1-4503-4660-3. DOI: 10.1145/3009837.3009872. URL: https://dl.acm.org/doi/10.1145/3009837.3009872 (visited on 04/08/2023).

Lindley, Sam, Conor McBride, and Craig McLaughlin (Oct. 3, 2017). *Do be do be do*. arXiv: 1611.09259[cs]. URL: http://arxiv.org/abs/1611.09259 (visited on 04/08/2023).

Moggi, Eugenio (1989a). *An Abstract View of Programming Languages*.

— (1989b). “Computational lambda-calculus and monads”. In: *[1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science*. [1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science. Pacific Grove, CA, USA: IEEE Comput. Soc. Press, pp. 14–23. ISBN: 978-0-8186-1954-0. DOI: 10.1109/LICS.1989.39155. URL: http://ieeexplore.ieee.org/document/39155/ (visited on 04/08/2023).

— (July 1991). “Notions of computation and monads”. In: *Information and Computation* 93.1, pp. 55–92. ISSN: 08905401. DOI: 10.1016/0890-5401(91)90052-4. URL: https://linkinghub.elsevier.com/retrieve/pii/0890540191900524 (visited on 01/12/2023).

Peyton Jones, Simon L. and Philip Wadler (1993). “Imperative functional programming”. In: *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL ’93*. the 20th ACM SIGPLAN-SIGACT symposium. Charleston, South Carolina, United States: ACM Press, pp. 71–84. ISBN: 978-0-89791-560-1. DOI: 10.1145/158511.158524. URL: http://portal.acm.org/citation.cfm?doid=158511.158524 (visited on 06/03/2023).

Plotkin, Gordon and John Power (2001). “Adequacy for Algebraic Effects”. In: *Foundations of Software Science and Computation Structures*. Ed. by Furio Honsell and Marino Miculan. Red. by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen. Vol. 2030. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 1–24. ISBN: 978-3-540-41864-1 978-3-540-45315-4. DOI: 10.1007/3-540-45315-6_1. URL: http://link.springer.com/10.1007/3-540-45315-6_1 (visited on 04/08/2023).

— (2003). “Algebraic Operations and Generic Effects”. In: *Applied Categorical Structures* 11.1, pp. 69–94. ISSN: 09272852. DOI: 10.1023/A:1023064908962. URL: http://link.springer.com/10.1023/A:1023064908962 (visited on 06/03/2023).

Plotkin, Gordon and Matija Pretnar (2009). “Handlers of Algebraic Effects”. In: *Programming Languages and Systems*. Ed. by Giuseppe Castagna. Vol. 5502. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 80–94. ISBN: 978-3-642-00589-3 978-3-642-00590-9. DOI: 10.1007/978-3-642-00590-9_7. URL: http://link.springer.com/10.1007/978-3-642-00590-9_7 (visited on 04/08/2023).

Schrijvers, Tom et al. (Aug. 8, 2019). “Monad transformers and modular algebraic effects: what binds them together”. In: *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*. ICFP ’19: ACM SIGPLAN International Conference on Functional Programming. Berlin Germany: ACM, pp. 98–113. ISBN: 978-1-4503-6813-1. DOI: 10.1145/3331545.3342595. URL: https://dl.acm.org/doi/10.1145/3331545.3342595 (visited on 06/11/2023).

Swierstra, Wouter (July 2008). “Data types à la carte”. In: *Journal of Functional Programming* 18.4. ISSN: 0956-7968, 1469-7653. DOI: 10.1017/S0956796808006758. URL: http://www.journals.cambridge.org/abstract_S0956796808006758 (visited on 06/11/2023).

Wu, Nicolas, Tom Schrijvers, and Ralf Hinze (June 10, 2014). *Effect Handlers in Scope*.

Xie, Ningning et al. (Oct. 31, 2022). “First-class names for effect handlers”. In: *Proceedings of the ACM on Programming Languages* 6 (OOPSLA2), pp. 30–59. ISSN: 2475-1421. DOI: `10.1145/3563289`. URL: `https://dl.acm.org/doi/10.1145/3563289` (visited on 06/13/2023).