Higher-Order Effects with Implicitly Resolved Elaborations

Version of June 13, 2023

Terts Diepraam

Higher-Order Effects with Implicitly Resolved Elaborations

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

TUDelft

Programming Languages Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
`www.ewi.tudelft.nl`

Cover picture: Random maze.

Higher-Order Effects with Implicitly Resolved Elaborations

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

Abstract

Preface

Preface here.

Terts Diepraam
Delft, the Netherlands
June 13, 2023

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In an idealized perspective on computation, programs and their subcomponents are *pure* functions. They take some input and for every set of inputs they return the same output, without interacting with other parts of the program or with the system. There is a certain elegance to this view: pure programs are relatively easy to reason about and analyse.

However, in practice, many programs are *impure*. These programs might interact with memory and the network, write to and read from the terminal. Additionally, programs might be clearer when they can be expressed with more complex control flow structures like exceptions and coroutines. We call these extensions on top of pure computation *effects* (Moggi 1989b).

Some programming languages, such as C and many of the languages it inspired, have opted to give the programmer unrestriced access to effectful operations. In particular, any part of the program can access memory, the filesystem and even allowed to use `goto` to jump to other parts outside of regular control flow. This unrestricted use has famously been criticized by Dijkstra (1968) and others. An open challenge in language design is then to design a language which allows for (limited) impure calculation, while retaining the attractive qualities of pure languages.

A promising aproach to striking this balance is the framework of algebraic effects by Plotkin and Power (2001) and extended by Plotkin and Pretnar (2009). Here, algebraic effects are only allowed within *handlers*, which have some, but still limited, control over the flow of the program via continuations. In recent years, some languages (e.g. Koka (Leijen 2017)) have been created with support for algebraic effects. However, there are some common patterns of effects that cannot be represented as algebraic effects, namely higher-order effects, that is, effects whose operations that take effectful computations as arguments.

To overcome this limitation, Bach Poulsen and Rest (2023) have extended algebraic effects with *hefty algebras*. They introduce *elaborations* in addition to handlers, which can encode higher-order effects.

In this thesis, we build on the work by Bach Poulsen and Rest (2023), applying the theory of hefty algebras as a first-class construct in a novel programming language called *Elaine*. Like Koka and other languages with support for algebraic effects, Elaine supports handlers. In addition, elaborations and higher-order effects are also supported.

We provide a novel reduction semantics and type system for a language with supporting elaborations. Furthermore, we introduce two transformations on Elaine programs. The first transformation is a type directed inference of elaborations, allowing the names of elaborations in use to be omitted. The second is a transformation from a program with higher-order effects and elaborations to a program with only algebraic effects. This transformation shows that elaborations can be added to existing languages and libraries for effects with relative ease.

In addition, we provide a full implementation of this language[1]. This implementation

[1]available at `https://github.com/tertsdiepraam/thesis/tree/main/elaine`

includes a parser, type checker, pretty printer, type checker and the two transformations described in this thesis.

Chapter 2

Background

While algebraic effects 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 this history.

Give definitions of effects, pure/impure, effectless/effectful

2.1 Motivation for Effects

First, let us pose a simple question: why study effects? As Moggi (1989b) notes, analyzing only pure computation leaves out many aspects of programs, such as side-effects, non-determinism and non-termination. Reasoning about the behaviour of a program then necessarily needs to include an analysis of effects.

Proving either the presence or absence of these properties is very valuable. For example, a deterministic, side-effect-less computation can be reliably cached. Without side-effects, computation can also be executed out of order without any observable difference. Hence, many sophisticated compilers rely on effect tracking to inform the optimizations they perform.Citation needed A theory of effectful computation can therefore help compiler engineers prove the correctness of their transformations.Citation needed Hence, a language with a strong formal model for effects is easier to compile or optimize than one without.

In addition, support for effects in the type system of a programming language could allow programmers to write stricter and safer APIs. In a language where the type system can reason about effects, the compiler can make certain guarantees beyond type checking only the input and output types of a function.

These guarantees fall into two categories: communicating presence and requiring absence. Communicating presence gives library authors the ability to tell the type system what effects their functions require. This informs users of libraries about what those functions can do (e.g. accessing the network or being able to run without terminating). This is often most useful when the effects are interaction with the environment,in which case effects can function as capabilities (Brachthäuser, Schuster, and Ostermann 2020).

A library author might also want to communicate that a function should not contain certain effects. For example, a hash function is generally understood to be deterministic and effectless, because it needs to be reproducible. However, this is often not guaranteed by the type system of mainstream programming languages.Citation needed

2.2 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 concered 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.

A limitation of treating effects as monads is that monads do not compose well; the composition of two monads is not itself a monad. A solution to this are *monad transformers* (Moggi 1989a), which are functors over monads that add operations to a monad. A simple 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.

2.3 Algebraic Effects

Introduce algebraic effects, effect rows, and handlers.

2.3.1 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 (2018).

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 operations 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 $\mathrm{Tree}_\Sigma(X)$ of *well-founded trees* over Σ generated by X. This set is defined inductively:

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

A Σ-term is a pair of a context X and a tree $t \in \mathrm{Tree}_\Sigma(X)$. We write a Σ-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 (Σ-equation)**.** A Σ-equation is a pair of Σ-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 Σ is unambiguous, we will omit the Σ 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 Σ_T and a collection $\mathcal{E}_\mathsf{T}$ of Σ_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 Σ is a

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

$$[\![op_i]\!]_I : P_i \times |I|_i^A \to |I|,$$

called an *operation.*

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

$$[\![t]\!]_I : |I|^X \to |I|.$$

This map is defined as

$$[\![\mathsf{return}\, x]\!]_I : \eta \mapsto \eta(x) \qquad [\![op_i(p,\kappa)]\!]_I : \eta \mapsto [\![op_i]\!]_I(p, \lambda a.[\![\kappa(a)]\!]_I(\eta)).$$

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

The *semantic bracket* $[\![]\!]_I$ 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. this true?

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

$$[\![l]\!] = [\![r]\!].$$

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

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

$$\phi \circ [\![op_i]\!]_L(p, \kappa) = [\![op_i]\!]_M(k, \phi \circ k).$$

That is, if ϕ commutes with operations. The models and the 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.

2.3.2 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 $op(p_1, \ldots, p_n)$ for some (effectful) operation op, with parameters $p_1, \ldots, p_n$.

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

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

We will add $\{\}$ around sequences 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

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

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

$$\kappa \ggg\!= \kappa' \quad \overset{\mathsf{def}}{=} \quad \{x \leftarrow \kappa;\ \kappa'(x)\}.$$

As an example, take the `State` effect. To use the 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`, returning the result of the `get`.

2.3.3 Effects as Algebraic Theories

Plotkin and Power (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`:

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

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:

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

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, non-termination, iteration, cooperative asynchronicity, traversal, input and output Citation needed. These effects are called *algebraic effects*.

As shown byPlotkin and Power (2003), an effect is algebraic if and only if it satisfies the *algebraicity property*, which can be expressed as follows:

$$op(m_1, \ldots, m_n) \ggg\!= \kappa \quad = \quad op(p, m_1 \ggg\!= \kappa, \ldots, m_n \ggg\!= \kappa).$$

In other words, all computation parameters must be *continuation-like*, that is, they are some computation followed by the continuation (Bach Poulsen and Rest 2023). This property is called the *algebraicity property* (Plotkin and Power 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

$$\texttt{local}(f, m) \ggg\!= \texttt{ask}() \quad \neq \quad \texttt{local}(f, m \ggg\!= \texttt{ask}()),$$

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 `throw`. The problem arises when we bind with an `throw` operation:

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

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.

2.3.4 Effect handlers

The distinction between effects which are and which are not algebraic has been described as the difference between *effect constructors* and *effect 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 subexpression.

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

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

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

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

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

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

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 (Bach Poulsen and Rest 2023).

2.4 Elaborations

Several solutions to the modularity have been proposed (Berg et al. 2021; Wu, Schrijvers, and Hinze 2014). Most recently, Bach Poulsen and Rest (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. After the elaboration, we can then handle the algebraic effects like we would for any algebraic effect.

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

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

which we can then apply to an expression with the elab keyword, similarly to handle:

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

This computation will evaluate to 15. One way to think about elaboration operations is as modular macros, which perform a syntactic substitution, based on the given elaboration.

Chapter 3

Related Work

- Koka
- Frank
- Eff
- etc.

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 implementation 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 allow for *effect row polymorphism*.

3.0.1 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 the connection between monad transformers and algebraic effects is very strong (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.

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

```
data Free f a
  = Pure 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

```
  | Do (f (Free f a))
```

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 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. As a result, most Haskell libraries for algebraic effecs require many GHC language extensions to provide an ergonomic interface. 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` 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
```

This signature is much 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.

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 signatures of `readfile` above, using the function and manipulating the output looks the same. 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`, but some higher-order effects are not supported.

any simple examples?

3.0.2 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 2017), Idris (Brady 2013) and Frank (Lindley, McBride, and

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

McLaughlin 2017), which are all specialized around effects. OCaml also gained support for effects (Sivaramakrishnan et al. 2021).

We can write the `readfile` signature and `firstline` function from before in Koka as follows:

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

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 not longer necessary in Koka.

The tail can be used to ensure the same effects across multiple functions. 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 and are explained in Chapter 4.

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 4

A Tour of Elaine

- Begin with examples
- Especially elaborations from HA paper
- Explain relation between Elaine and Hefty Algebras
- Spec
  - Syntax
  - Reduction semantics
  - Typing rules & row equivalence
  - Type checker: unification rules mostly

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 resulution and compilation of elaborations, which are explained in Chapter 6 and **??**, 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 5. More example programs are available online[1].

4.1 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 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.

[1]`https://github.com/tertsdiepraam/thesis/tree/main/elaine/examples`

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.

```
# 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, x) {
   let text = concat(show_int(x), " is ");
   let answer = f(x);
   concat(text, answer)
};

let main = give_answer(square_is_even, 4);
```

4.2 Algebraic Effects

The programs in the previous section are all pure and contain no effects. Like the languages discussed in Chapter 3, 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 4.2.2)

We will be using the following effects throughout this section.

Add corresponding monads?

```
# Defines a single operation to get an implicit variable
effect Val {
   val() Int
}

# Exits the current handle
```

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

# Allows for a mutable state via a get and a set operation
effect State {
   get() Int
   set(Int) ()
}
```

4.2.1 Effect Rows

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).

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 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) other set of 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 4.4.1.

4.2.2 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 `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.

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

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

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.

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

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

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 of the **`handle`**.

Alternatively, we can define a handler that defines a default value for failing expressions.

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

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

```
  }
};

let main = handle[hAbort] safe_div(5, 0);
```

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

Even for small handlers I need custom data types

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

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.[2]

```
let hAbort = handler Abort {
  return(x) { x }
  abort() { resume(()) }
};
```

4.3 Higher-Order Effects in Elaine

4.4 Specification of Elaine

4.4.1 Type Checker

Unification of Effect Rows

Talk about (Leijen 2005).

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 the language 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:

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

[2]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.

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.

Chapter 5

Elaine Specification

5.1 Syntax definition

$$
\begin{aligned}
\text{program } p &::= m \ldots m \\
\text{module } m &::= \texttt{mod}\ x\ \{d \ldots d\} \\
\\
\text{declaration } d &::= \texttt{pub}\ d' \ \mid\ d' \\
d' &::= \texttt{let}\ x = e; \\
&\mid \texttt{import}\ x; \\
&\mid \texttt{effect}\ \phi\ \{s, \ldots, s\} \\
&\mid \texttt{type}\ x\ \{s, \ldots, s\} \\
\text{expression } e &::= x \\
&\mid () \ \mid\ \texttt{true} \ \mid\ \texttt{false} \\
&\mid \texttt{fn}(x : T, \ldots, x : T)\ T\ \{e\} \\
&\mid \texttt{if}\ e\ \{e\}\ \texttt{else}\ \{e\} \\
&\mid e(e, \ldots, e) \\
&\mid x!(e, \ldots, e) \\
&\mid \texttt{handler}\ \{return(x)\{e\}, o, \ldots, o\} \\
&\mid \texttt{handle}[e]\ e \\
&\mid \texttt{elaboration}\ x! \to \Delta\ \{o, \ldots, o\} \\
&\mid \texttt{elab}[e]\ e \\
&\mid \texttt{elab}\ e \\
&\mid \texttt{let}\ x = e;\ e \\
&\mid e;\ e \\
&\mid \{e\} \\
\\
\text{signature } s &::= x(T, \ldots, T)\ T \\
\text{effect clause } o &::= x(x, \ldots, x)\ \{e\} \\
\\
\text{type scheme } \sigma &::= T \ \mid\ \forall \alpha.\sigma \\
\text{type } T &::= \Delta\ \tau \\
\text{value type } \tau &::= x \ \mid\ () \ \mid\ \texttt{Bool} \\
&\mid (T, \ldots, T) \to T
\end{aligned}
$$

$$
\begin{aligned}
&\mid \texttt{handler}\ x\ \tau\ \tau\\
&\mid \texttt{elaboration}\ x!\ \Delta\\
\text{effect row } \Delta &::= \langle\rangle \ \mid\ x \ \mid\ \langle \phi|\Delta\rangle\\
\text{effect } \phi &::= x \ \mid\ x!
\end{aligned}
$$

5.2 Typing judgments

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

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

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.

5.2.1 Effect row semantics

We treat effect rows as multisets. That means that the row $\langle A, B, B, C\rangle$ is simply the multiset $\{A, B, B, C\}$. The | symbol signifies extension of the effect row with another (possibly arbitrary) effect row. The order of the effects is insignificant, though the multiplicity 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 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:

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

5.2.2 Type inference

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

Koka requires an empty effect row. Why?

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

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

5.2.3 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.

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

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

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

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

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

5.2.4 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 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.

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

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

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

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

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

5.2.5 First-Order 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 the input and output type.

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

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

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

5.2.6 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.

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

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.

It is not possible to elab

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

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

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

5.3 Desugaring

Fold over the syntax tree with the following operation:

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

5.4 Elaboration resolution

5.5 Semantics

5.5.1 Reduction contexts

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

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

5.5.2 Reduction rules

$$
\begin{array}{rcl}
c(v_1, \ldots, v_n) & \longrightarrow & \delta(c, v_1, \ldots, v_n) \\
 & & \qquad \text{if } \delta(c, v_1, \ldots, v_n) \text{ defined} \\
(\lambda x_1, \ldots, x_n . e)(v_1, \ldots, v_n) & \longrightarrow & e[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] \\
\texttt{if true } \{e_1\} \texttt{ else } \{e_2\} & \longrightarrow & e_1 \\
\texttt{if false } \{e_1\} \texttt{ else } \{e_2\} & \longrightarrow & e_2 \\
 & & \\
\texttt{handle}[h]\ v & \longrightarrow & e[x \mapsto v] \\
 & & \qquad \text{where } \texttt{return}(x)\{e\} \in H \\
\texttt{handle}[h]\ X_{op}[op(v_1, \ldots, v_n)] & \longrightarrow & e[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n, resume \mapsto k] \\
 & & \qquad \text{where } op(x_1, \ldots, x_n)\{e\} \in h \\
 & & \qquad\qquad\quad k = \lambda y \ . \ \texttt{handle}[h]\ X_{op}[y] \\
\texttt{elab}[\epsilon]\ v & \longrightarrow & v \\
\texttt{elab}[\epsilon]\ X_{op!}[op!(e_1, \ldots, e_n)] & \longrightarrow & \texttt{elab}[\epsilon]\ X_{op!}[e[x_1 \mapsto e_1, \ldots, x_n \mapsto e_n]] \\
 & & \qquad \text{where } op!(x_1, \ldots, x_n)\{e\} \in \epsilon
\end{array}
$$

Chapter 6

Elaboration Resolution

- How it works
- Why it works

Chapter 7

Elaboration Compilation

- How it works
- Why it works
- Preserves well-typedness
- Why syntactic substitution of elaborations with implementations does not work.
- Consequences for research on higher-order effects in general.

Chapter 8

Conclusion

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).

Berg, Birthe van den 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).

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

— (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).

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).

Sivaramakrishnan, K. C. et al. (June 19, 2021). "Retrofitting Effect Handlers onto OCaml". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pp. 206–221. DOI: `10.1145/3453483.3454039`. arXiv: `2104.00250[cs]`. URL: `http://arxiv.org/abs/2104.00250` (visited on 04/08/2023).

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).