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

*Version of September 6, 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 |

# Contents

# Chapter 1

# Introduction

**TODO**: Start: effects are a thing

**TODO**: Examples in Koka throughout!

In many programming languages, computations are allowed to have *effects*. This means that they can do things besides producing output and interact with their environment. It might, for instance, read or modify a global variable, write to a file, throw an exception or even exit the program altogether. These are all examples of *effects*.

Historically, programming languages have supported effects in many different ways. 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 in program with the `goto` keyword, which has famously been criticized by Dijkstra (1968). This core of his argument is that `goto` breaks the structure of the code. The programmer then has to trace the execution of the program in their mind in order to understand it. The same reasoning extends to other effects too: the more effects a function is allowed to exhibit, the harder it becomes to reason about.

The "anything goes" approach to effects therefore puts a large burden of ensuring correct behaviour of effects on the programmer. If the language cannot give any guarantees about what a function can do, the programmer has to check instead. Say, for instance, that a function somewhere in the code is modifying some global variable to some invalid value. This can then make entirely different parts of the program behave incorrectly. The programmer tasked with debugging this issue then has to examine the program as a whole to find where this modification takes place, if they even realize that it is set incorrectly. In languages where this is possible, effectful operations therefore limit our ability to split the code into chucks to be examined separately.

A solution is to treat some effects in a more structured manner. For example, instead of allowing `goto`, a language might provide exceptions. In a language like Java, the exceptions are then also part of the type system, so that it is easier to track which functions are allowed to throw exceptions. However, this means that any effect must be backed by the language. New effects cannot be created without adding a new feature to the language. This means that the support for various effects is always limited.

In contrast, languages adhering to the functional programming paradigm disallow (with some exceptions) effectful operations alltogether. Here, all functions are *pure*, meaning that they are functions in the mathematical sense: only a mapping from inputs to output. Such a function is *referentially transparent*, meaing that it always returns identical outputs for identical inputs. They also do not interact with their environment. By dictating that all functions are pure, a type signature of a function becomes almost a full specification of what the function can do. In that sense, the return type is an all-encompassing description of what a function might do.

**TODO**: Example of manual state: checking balanced parentheses? Fibonacci? Reverse polish notation? Wadler & Peyton Jones use IO, that's nice I guess.

**TODO**: Pure means that the effect must be encoded in the return type of the function. We want types to be "all-encompassing", "reliable".

FEEDBACK: vague

Yet this rule is quite limiting, since effects are often an important part of a computation. For instance, if we want to keep some mutable state `a` in a Haskell program, we have to encode that state in the inputs and outputs of the program. Manually threading the state through the program quickly becomes laborious in larger programs. The same goes for encodings of other effects. A more practical method of dealing with effectful operations in functional languages is through the use of *monads* (Peyton Jones and Wadler 1993; Wadler 1992).

Monads were introduced as a mathematical model for effectful computation by Moggi (1991). A function returning a monad is not fully executed. Instead, it is evaluated until the first effectful operation is encountered. This partially evaluated result is only further evaluated when it is passed to a *handler*. This handler decides what to do with this result and can resume the computation, which will again evaluate the until the next effectful operation and the cycle repeats. If we then wish to have some state in our Haskell program, we have to wrap all our stateful functions in the `State` monad and pass it to the `runState` handler.

FEEDBACK: modularity is about syntax AND semantics. Reason about individual effects and compose them. Modular handlers with different implementations. And we can create new "syntax'

This split between the procedure and the handler provides some modularity. We can swap out the standard `runState` handler for some other handler. We might for instance write a handler that does not just yield the final value of the state, but a list containing the history of all values that the state has been set to. Or, we create a handler where every `get` operation increases the value of the state by one, such that every `get` yields a unique value. This is all possible without changing the code using the `State` monad.

The limitations of the monad approach become apparent when we look at procedures that use multiple effects. The problem is that the composition of two monads does not yield a monad. This is a limitation that can be worked around with *monad transformers*, which can be applied to monads to yield a new monad. However, this yields complicated types which have to be taken into account while programming. Operations might need to be lifted from between monads. Additionally, monad transformers need quadractic implementation stuffs. Therefore, working with monad transformers is still quite laborious.

Therefore, our goal is to design a programming language in which the function signature forms the full specification of all effects, which also allows new effects to be declared, while also being easy to work with. Luckily, the theory of *algebraic effects* fulfills these requirements. First, it removes the distinction between monads and monad transformers. Second, it ignores the order of effects in types. 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*. The operations 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.

**TODO**: ADD EXAMPLES

As it turns out, however, not all effects are algebraic. *Higher-order effects* are effects with operations that take effectful computations as arguments that do not behave like continuations. The issue is that the handlers need be able to handle the effect and then let the rest of the computation evaluate, but that is not sufficient for higher-order effects.

Several extensions and modifications to algebraic effects have been proposed to accomodate for higher-order effects (van den Berg et al. 2021; Wu, Schrijvers, and Hinze 2014). One such extension is *hefty algebras* by Bach Poulsen and van der Rest (2023), which introduces elaborations to implement higher-order effects. This is a mechanism to define higher-order effects by defining a translation into a program with only algebraic effects. This means that evaluation of higher-order effects is a two-step process: first higher-order effects are elaborated into algebraic effects, which are then evaluated. Like handlers, elaborations are modular and it is possible to define multiple elaborations for a single effect.

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, elaborations 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.

Additionally, we give 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. Elaine opens up exploration for programming languages with higher-order effects. While not a viable general purpose language in its own right, it can serve as inspiration for future languages.

## 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 (Appendix B.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

## Algebraic Effects

Elaine is based on the theory of hefty algebras, which is an extension of the theory of algebraic effects. Hence, the theory of algebraic effects also applies. In this chapter, we give an intoduction to algebraic effects. In the next chapter, we discuss its limitations regarding higher-order effects and describe how hefty algebras overcome those limitations.

This chapter builds up to a fairly established model for programming languages, such as Koka. Readers familiar with algebraic effects and Koka may therefore want to skip this chapter.

### 2.1 Monads

We will build up the notion of algebraic effects from monads. Monads are an abstraction over effectful computation commonly used in functional programming.

While many descriptions of monads using catagory theory and analogies are possible, for our purposes, a monad is a type constructor `m` and two functions: `return` and `>>=`, with the latter pronounced "bind". In Haskell, this concept is easily encoded in a type class, which is listed below, along with the definition of a funtion `>>`.

```
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b

(>>) :: Monad m => m a -> m b -> m b
a >> b = a >>= \_ -> b
```

This class tells us that we can construct a value of `m a` for any type `a` and for any monad `m` using `return`. Additionally we can compose two monads using `>>=` by providing an `m a` and a function from `a` to `m b`.

To explain how effectful operations can be encoded with this, we can look at a simple example: the `Maybe` monad. Our goal with this monad is to create an "abort" effect, where the computation stops an returns immediately once `Nothing` is encountered.

```
data Maybe a
  = Just a
  | Nothing

class Monad Maybe where
  return = Just

  Just a  >>= k = k a
  Nothing >>= k = Nothing
```

With this definition, we can chain functions returning `Maybe`. For example, we can define a `head` function that returns the first element of a list if it is non-empty and `Nothing` otherwise. We can also define a division function which checks that the divisor is non-zero. These functions can then be composed using `>>=`.

```
head :: [a] -> Maybe a
head (x:xs) = Just x
head _ = Nothing

safeDiv :: Int -> Int -> Maybe Int
safeDiv _ 0 = Nothing
safeDiv x y = div x y

main = do
  print $ head []      >>= safeDiv 10 -- -> Nothing
  print $ head [0,1,2] >>= safeDiv 10 -- -> Nothing
  print $ head [2,3,4] >>= safeDiv 10 -- -> Just 5
```

A more involved example is the `State` monad. If we were to keep track of state manually a function that modifies state would need to take some state `s` and an argument and return a new values for the state. Therefore if a function `foo` normally is a function with type `a -> b`, it would need to have the type `a -> s -> (s, b)` to modify state. Then we need to ensure that we update the state with the modified value. For example, if the function is called multiple times, the code would look something like the code before.

```
-- Increment the state by `a` and return the old state
inc :: Int -> Int -> (Int, Int)
inc a s = (s + a, s)

multipleIncs :: Int -> (Int, Int)
multipleIncs s = let
  (s' , _) = inc 5 s
  (s'', _) = inc 6 s'
  in inc 7 s''
```

The program becomes verbose and repetitive as a result. If we look at the signature `a -> s -> (s, b)`, we can see that there is a possibility for abstraction here: we can abstract over the pattern of `s -> (s, b)`. Or definition of the `State` type then becomes:

```
newtype State s a = State (s -> (s, a))

-- so that the inc function becomes:
inc :: Int -> State Int Int
inc a = State $ \s -> (s + a, s)
```

That might look like a step backwards at first, but we can now implement the monad functions for `State s` to allow us to compose functions returning the `State` type. Additionally we define some the `get` and `put` operations, which are the basic building blocks we can use to build more complex operations.

```
instance Monad (State s) where
  return = State (, a)

  State fa >>= k = State $ \s ->
    let (s', a) = fa s
        State fb = k a
```

```haskell
      in fb s'

get :: State s s
get = State $ \s -> (s, s)

put :: s -> State s ()
put = State $ \s -> (s, ())
```

For convenience, we also define `runState` to allow us to provide an initial state and evaluate the entire computation.

```haskell
runState :: s -> State s a -> (s, a)
runState init (State s) = s init
```

Using the bind operator, we can then reduce our `bar` function to the following.

```haskell
inc :: Int -> State Int Int
inc x = get >>= \s -> put (s + x) >>= return s

bar :: State Int Int
bar = inc 5 >> inc 6 >> inc 7

main = print $ runState 0 bar
```

This is the power of monads, they allow us to abstract the effectful operations away, while also signalling the effects that a function requires in the return type. In the final example, we do not have to think about how the `State` monad works anymore, but only use the `get` and `put` operations to build complex computations.

## 2.2 Composition with the Free Monad

A limitation of the monads above is that they cannot be composed: it is not possible to make a computation with them that uses both `State` and **`Maybe`** at the same time. One solution to this is to used monad transformers, as explained in Section 7.1. Another solution is to use the *free monad*.

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

instance Functor f => Monad (Free f) where
  return = Pure

  Pure x >>= f = f x
  Do g >>= f = Do $ fmap (>>= f) g
```

There is only one thing required to build a state monad using the free monad: `State` needs to be a functor, meaning that it needs an implementation of `fmap`. The implementations of **`return`** and `>>=` are then provided by the free monad.

```haskell
data State s a = State (s -> (s, a))

instance Functor (State s) where
  fmap f (State r) = State $ \s -> let (s', a) = r s in (s', f a)
```

The `Free (State s)` monad now works just like the `State s` monad from the previous section. Similarly we can apply the free monad to **`Maybe`**. However, the **`Just`** constructor of the **Maybe** is already covered by the `Pure` constructor of the free monad, so **Maybe** can be simplified to a single constructor. We call this simplified type `Abort`.

```
data Abort a = Abort

instance Functor Maybe where
  fmap _ Abort  = Abort
```

The trick to composing effects is that functors, in contrast with monads, can be meaningfully composed. We define a type-level operator `+`, which can be thought of as **Either** for functors.

```
infixr 6 +
data (f + g) a = L (f a) | R (g a)
  deriving Functor
```

This means that for any set of functors `f1`, ..., `fN`, we can construct a monad `Free (f1 + ... + fN)`. However, we have no way to use any of the effect operations for this functor. For example, if we have `Free (State s + Abort)`, how would we use the `get` operation? The solution is to give a definition for `get` for the free monad if and only if `State` is one of the composed functors. We do this with a typeclass relation `<`, which defines an injection from a functor `f` to any composed functor `g` that contains `f`. We can use this injection to define `get`, `put` and `abort`.

```
class f < g where
  inj :: f k -> g k

instance f < f where inj = id
instance f < (f + g) where inj = L
instance f < h => f < (g + h) where inj = R . inj

get :: State s < f => Free f s
get = Do $ inj $ Pure <$> State (\s -> (s,s))

put  :: State s < f => s -> Free f ()
put s = Do $ inj $ Pure <$> State (const (s, ()))

abort :: Abort < f => Free f ()
abort = Do $ inj $ Abort
```

```
fold :: Functor f => (a -> b) -> (f b -> b) -> Free f a -> b
fold gen _   (Pure x) = gen x
fold gen alg (Op f)   = alg (fmap (fold gen alg) f)

handle :: (a -> Free f' b) -> (f (Free f' b) -> Free f' b) -> Free (f + f') a -> Free
handle ret hndl = fold ret (\case
    L x -> hndl x
    R x -> Do x
)

handleAbort = handle (Pure . Just) (\Abort -> Nothing)
```

**TODO**: State handler

This finally allows us to use the `Abort` and `State` monads together, while providing a handler per functor. While the plumbing needed for a free monad is encoding, it is worth considering what it provides. First, we can combine multiple functors in our type signatures. Second, we can define operations that work for any effect composition that contains an effect. Third, we can provide modular handlers that handle a single effect from the composed functors. Any effect we define in this way is automatically compatible with all the other effects.

## 2.3 Algebraic Effects

The term “algebraic” comes from the fact that this method works for effects that can be described as algebraic theories when encoded in a continuation-passing style (Plotkin and Power 2001). This can only be done for effects that satisfy the *algebraicity property*.

## 2.4 Building a Language with Algebraic Effects

An algebraic effect system is an effect system based on the ideas in the previous section. In fact, the previous section contains an implementation of an algebraic effect system. We can use these concepts to then design a language that integrates this encoding in the language directly. The advantage of that is that all the plumbing can be hidden and that even more convenient syntax for using effects can be introduced.

At the core of such languages is the following rule: all functions return the free monad. That means that a function `a -> b` is not allowed, but `a -> Free f b` is allowed. Because that is a bit verbose, we can introduce a shorthand: `a -> f b`. We will no longer refer to `f` as a functor, but as an *effect row* and its elements as *effects*. Because we change the name, we will also generally use an `e` instead of an `f` to denote an effect row. So, the function type `a -> e b` should be read as: this function takes an `a` and returns `b` with effect row `e`.

Instead of using type-level operators, we can introduce special syntax for effect rows, too. Following the lead of Koka citation needed, we will write effect rows as

`<e1,e2,...,eN|es>`,

where the `|es` part is optional. Effects `e1` to `eN` are the explicit effects and `es` is the tail; a variable representing the effect row with which this effect row can be extended. In the type system, we are then allowed to use different orders of effects interchangeably. spelling

The use of algebraic effects is discussed more in later chapters, but we give a small example here in Koka. We define the `abort` and `state` effects with their corresponding handlers.

**TODO**: Fill this in, but don't use too many weird koka features

```
hello, world!
```

## 2.5 Effect Handlers

**TODO**: This is trash atm

FEEDBACK: This needs to be expanded.

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:

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

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

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

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

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

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 (Wu, Schrijvers, and Hinze 2014).

# Chapter 3

# Higher-Order Effects

## 3.1 Higher-Order Effects

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

**TODO**: Format nicely

```
let x = local(f, m); ask()   ≠   local(f, {let x = m; 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:

**TODO**: Format nicely

```
let x = catch(m1, m2); throw()   ≠   catch({let x = m1; throw()}, {let x = m2;throw()}).
```

On the left-hand side, `m2` will not be executed if `m1` does not throw, while on the right-hand side, `m2` will always get executed. The two sides therefore have different semantics and hence the `catch` operation is not algebraic.

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.

## 3.2 Elaborations

**TODO**: Move this somewhere else

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.

FEEDBACK: This needs explanation

FEEDBACK: which definition?

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:

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

FEEDBACK: language we're working in is unclear

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

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

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

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

FEEDBACK: give example

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

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

# Chapter 4

# 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 Appendix B. Additional example programs are available in the artefact accompanying this thesis, the contents of which are detailed in Section 1.2 and in Appendix A.

## 4.1 Basics

As is tradition with introductions to programming languages, we have to start with a program that shows the string `"Hello, world!"`:

```
# The value bound to main is the return value of the program
let main = "Hello, world!";
```

There are several aspects of Elaine that this example highlights. Comments start with `#` and continue until the end of the line. We bind variables with the **`let`** keyword. The `main` variable is required and is the value printed at the end of execution. In contrast with other languages, `main` is not a function in `Elaine`. Note too that statements are required to end with a semicolon.

In addition to strings, Elaine features integers and booleans. The latter is expressed the **`false`** and **`true`** keywords. To operate on these types, we need functions to perform the operations. By default, there are no functions in scope, however, we can bring them in scope by importing the functions from the `std` module with the **`use`** keyword. For example, we can write a program that computes $5 \cdot 2 + 3$:

```
use std;
let main = add(mul(5, 2), 3);
```

The `std` module contains functions for boolean and numeric arithmetic, comparison of values and more. The full list of functions is given in Appendix B.6. To show off some more functions, below is a program that returns a boolean indicating whether $2^5$ is greater than $-(25 \cdot -30)$, which is true. Note that `-` is allowed as part of an integer literal, but not as an operator.

```
use std;
let main = gt(
    pow(2, 5),
    neg(mul(25, -30)),
);
```

Bindings can be used to split up a computation, both at the top-level and within braces, which are used to group sequential expressions. A sequence of expressions evaluates to the last expression. Expressions are only allowed to contain variables that have been defined above the expression, so the order of bindings is significant. This rule also disallows recursion. Below is the same comparison written with some bindings.

```
use std;
let a = pow(2, 5);
let main = {
    let b = mul(25, -30);
    gt(a, neg(b))
};
```

Functions are defined with **`fn`**, followed by a list of arguments and a function body. Unlike Haskell, functions are called with parentheses.

```
use std;
let double = fn(x) {
    mul(2, x)
};
let square = fn(x) {
    mul(x, x) # or pow(x, 2)
};
let main = double(square(8));
```

Tuples are comma-separated lists of expressions surrounded with `()`. Tuples have a fixed length and can have elements of different types.

```
let main = (9, "hello");
```

Additionally, Elaine features **`if`** expressions. The language does not support recursion or any othr looping construct. Listing 4.1 contains a program that uses the basic features of Elaine and prints whether the square of 4 is even or odd.

## 4.2 Types

Elaine is strongly typed with 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 primitive types are `String`, `Bool` and `Int` for strings, booleans and integers respectively. We can specify the types for let bindings, function arguments and return types.

```
let x: Int = 5;       # ok!
let x: String = 5;    # type error!

let f = fn(x: Int) Int { mul(3, x) };
let y = fn("Hello");  # type error!
```

We also could have written the type of the function as the type for the let binding. The type for a function is written like a function definition, without parameter names and body.

```
let f: fn(Int) Int = fn(x) { mul(3, x) };
```

Type parameters start with a lowercase letter. They do not need to be declared explicitly, much like in Haskell.

```
let f = fn(x: a) (a, Int) {
    (x, 5)
```

Listing 4.1: 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);
```

```
};
let y = f("hello");
let z = f(5);
```

## 4.3 Algebraic Data Types

However, these primitive types do not allow us to write very complex programs. To do that, we need to be able to define our own data types. That is what the **`type`** construct is for. It is analogous to Koka's **`type`**, Haskell's **`data`** or Rust's **`enum`** construct.

A type consists of a list of constructors each with a list of parameters. These constructors can be used as functions. A type can have type parameters, which are declared with `[]` after the type name. It is not possible to put constraints on type parameters.

Data types can be deconstructed with the **`match`** construct. The **`match`** construct looks like Rust's **`match`** or Haskell's **`case`**, but is more limited. It can only be used for custom data types and only matches on the outer constructor. For example, it is not possible to match on `Just(5)`, but only on `Just(x)`.

```
use std;

```

```
type Maybe[a] {
    Just(a),
    Nothing(),
}

let safe_div = fn(x, y) Maybe[Int] {
    if eq(y, 0) {
        Nothing
    } else {
        Just(div(x, y))
    }
};

let main = match safe_div(5, 0) {
    Just(x) => show_int(x),
    Nothing => "Division by zero!",
};
```

Since the `Maybe` type is very common, it is provided in the standard library under the `maybe` module.

Data types are allowed to be recursive. For example, we can define a `List` type as it is often defined in functional languages with a `Cons` and a `Nil` constructor.

```
type List[a] {
    Cons(a, List[a]),
    Nil(),
}

let list: List[Int] = Cons(1, Cons(2, Nil()));
```

The `List` type is also provided in the standard library in the `list` module. If that module is in scope there is also some syntactic sugar for lists: we can write a list with brackets and comma-separated expressions like `[1, 2, 3]`.

## 4.4 Recursion, Loops and Lists

The let bindings in the previous sections are not allowed to be recursive. In general, let bindings can only references values that have been defined before the binding itself. However, recursion or some other looping construct is necessary for many programs. Therefore, Elaine has a special syntax for recursive definitions: **`let rec`**.

Let bindings with **`rec`** definitions are desugared into the Y combinator. However, it is impossible to write the Y combinator manually, because it would have an infinite type. The type checker therefore has special case for recursive definitions.

An example of a recursive function is the `factorial` function listed below.

```
use std;

let rec factorial = fn(x: Int) {
    if eq(x, 0) {
        1
    } else {
        mul(x, factorial(sub(x,1)))
    }
};
```

A word of caution: Elaine has no guards against unbounded recursion of functions or even recursive expressions. For example, these statements are valid according to the Elaine type checker, but will cause infinite recursion when evaluated, until the interpreter runs out of memory:

```
# Warning: these declarations will not halt!
let rec f = fn(x) { f(x) };
let rec x = x;
```

Using recursive definitions, we can build functions like `map`, `foldl` and `foldr` to operate on our previously defined `List` type. The implementation for `map` might look like the listing below.

```
let rec map = fn(f: f(a) b, list: List[a]) List[b] {
    match list {
        Cons(a, as) => Cons(f(a), map(f, list)),
        Nil() => Nil(),
    }
};

let doubled = map(fn(x) { mul(2, x) }, [1, 2, 3]); # -> [2, 4, 6]
```

Note that, in contrast with Haskell, Elaine evaluates `map` eagerly; there is no lazy evaluation.

The `list` module provides the most common operations on lists. Such `head`, `concat_list`, `range`, `map`, `foldl` and `foldr`. It also provides a `sum` function for lists of integers and a `join` function for lists of strings.

## 4.5 Algebraic Effects

The programs in the previous sections are all pure and contain no effects. While a monadic approach is possible, Elaine provides first class support for algebraic effects and effect handlers to make working with effects more ergonomic. The design of effects in Elaine is heavily inspired by Koka citation needed.

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 4.5.1).

Figure 4.1 lists examples of effect declarations for the `Abort`, `Ask`, `State` and `Write` effects. We will refer to those declarations throughout this section. For the listings in this section, one can assume that these declarations are used. The `Abort` effect is meant to exit the computation. `Ask` provides some integer value to the computation, much like a global constant. `State` corresponds to the `State` monad in Haskell. Finally, `Write` allows us to write some string value somewhere. We will be using this to provide a substitute for writing to standard output.

### 4.5.1 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.

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

```
effect Ask {
    ask() Int,
}
```

```
effect State {
    get() Int,
    put(Int) (),
}
```

```
effect Write {
    write(String) (),
}
```

Figure 4.1: Examples of algebraic effect declarations for some simple effects.

For example, if we want to use an effect to provide an implicit value, we can make an effect `Ask` and a corresponding handler, which `resume`s execution with some values. The `resume` function represents the continuation of the program after the operation. The simplest handler for `Ask` we can write is one which yields some constant value.

```
let hAsk = handler { ask() { resume(10) } };

let main = handle[hAsk] add(ask(), ask()); # evaluates to 20
```

Of course, it would be cumbersome to write a separate handler for every value we would like to provide. 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 hAsk = fn(v: Int) {
    handler { ask() { resume(v) } }
};

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

The true power of algebraic effects, however, lies in the fact that we can also write a handler with entirely different behaviour, without modifying the computation. For example, we can creat a stateful handler which increments the value returned by `ask` on every call to provide unique identifiers. The program below will return 3, because the first `ask` call returns 1 and the second returns 2. This is accomplished in a very similar manner to the `State` monad.

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. 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()`. If the computation returns, the optional `return` arm of the handler will be applied. In the code below, this wraps the returned value in a `Just`. All arms of a handler must have the same return type.

```
let hAsk = handler {
    return(x) { fn(s: Int) { x } }
    ask() {
        fn(s: Int) {
            let f = resume(s);
            f(add(s, 1))
        }
    }
};

let c = handle[hAsk] add(ask(), ask());
let main = c(1);
```

```
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 the computation in case it aborts. This is more convenient that the first handler if the `abort` case should alwasy become

Just like we can ignore the continuation, we can also call it multiple times, which is useful for non-determinism and logic programming. In the listing below, the `Twice` effect is introduced, which calls its continuation twice. Combining that with the `State` effect as previously defined, the `put` operation is called twice, incrementing the initial state 3 by two, yielding a final result of 5. Admittedly, this example is a bit contrived. A more useful application of this technique can be found in Appendix A.1, which contains the full code for a very naive SAT solver in Elaine, using multiple continuations.

```
effect Twice {
    twice() ()
}

let hTwice = handler {
    twice() {
        resume(());
        resume(()))
    }
}

let main = {
```

```
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),
);
```

```
    let a = handle[hState] handle[hTwice] {
        twice();
        put(add(get(), 1));
        get()
    };
    a(3)
};
```

### 4.5.2 Effect Rows

All types in Elaine have an effect row. Up to this point, we have omitted the effect rows. This works because effect rows are inferred by the type checker. Effect rows represent the set of effects that need to be handled to obtain the value in a computation. 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 the `Ask` and `State` effects has type `<Ask,State> Int` or, equivalently `<State,Ask> Int`. The order of effects in effect rows is irrelevant. However, the multiplicity is important, that is, the effect rows `<State,State>` and `<State>` 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 denoted 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 called a row without extension *closed* and a row with extension *open*.

| | | | | |
|---|---|---|---|---|
| `<A>` | ∪ | `<>` | → | `<A>` |
| `<A>` | ∪ | `<A>` | → | `<A>` |
| `<A>` | ∪ | `<B>` | → | `<A,B>` |
| `<A,B>` | ∪ | `<B,A>` | → | `<A,B>` |
| `<A,A>` | ∪ | `<A>` | → | `<A,A>` |
| `<A,B|e>` | ∪ | `<C>` | → | `<A,B,C|e'>` |
| `<A|e>` | ∪ | `<B|e>` | → | `# error!` |
| `<A|e1>` | ∪ | `<B|e2>` | → | `<A,B|e3>` |

Table 4.1: Examples of effect row unification.

Like types, effect rows are unified in the type checker. For unification any closed row is first opened by introducing a new expansion variable. Then unification solves for the equation

`<A1,...,AN|e> = <B1,...,BM|f>`,

for `e` and `f`. To do so, a fresh variabled `g` is introduced which represents the intersection of `e` and `f`. The unified row then becomes `<A1,...,AN,B1,...,BN|g>`. Table 4.1 provides some more examples of effect row unification. The full procedure for unification is detailed in Appendix B.2.

We can use extensions to ensure equivalence between effect rows without specifying the full rows. 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() }
}
```

When an effect is handled, it is removed from the effect row. The `main` binding is required to have an empty effect row, which means that all effects in the program need to be handled. Therefore, to use the `abort_on_false` function defined above, it needs to be called from within a handler.

```
let main: <> Maybe[()] = handle[hAbort] {
    abort_on_false(fn() { false })
};
```

## 4.6 Effect-agnostic Functions

Recall the definition of `map` in Section 4.4, which was written without any effects in its signature. Adding the effects yields the following definition.

```
pub let rec map = fn(f: fn(a) <|e> b, l: List[a]) <|e> List[b] {
    match l {
        Nil() => Nil(),
        Cons(x, xs) => Cons(f(x), map(f, xs)),
    }
};
```

Note that the parameter `f` and `map` use the same effect row variable `e`. This means that `map` has the same effect row as `f` for any effect row that `f` might have, including the empty effect row. This makes `map` quite powerful, because it can be applied in many situations.

```
let pure_doubled = map(fn(x) { mul(2, x) }, [1,2,3]);
let ask_added = handle[hAsk(5)] map(fn(x) { add(ask() x) }, [1,2,3]);
```

If we were two write the same expressions in Haskell instead, we would need two different implementations of `map`: one for applying pure functions (**`map`**) and another for applying monadic functions (**`mapM`**). Our definition of `map` is therefore more general than Haskell's **`map`** function. The same reasoning can be applied to other functions like `foldl` and `foldr` or indeed any higher-order function.

## 4.7 Higher-Order Effects

Higher-order effects in Elaine are supported via elaborations, as proposed by Bach Poulsen and van der Rest (2023). In this framework, higher-order effects are elaborated into a computation using only algebraic effects. They are not handled directly. This means that we cannot write handlers for them as we did for algebraic effects in the previous section.

To distinguish higher-order effects and operations from algebraic effects and operations, we write them with a `!` suffix. For example, a higher-order `Exception!` effect is written `Exception!`, and its `catch` operations is written `catch!`.

Higher-order effects are treated exactly like algebraic effects in the effect rows. The order of effects still does not matter and we can create effect rows with arbitrary combinations of algebraic and higher-order effects.

The elaborated 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.

An illustrative example of this feature is the `Reader` effect with a `local` operation, shown in Appendix A.2. This effect enhances the previously introduced `Ask` effect with a `local` operation that modifies the value returned by `ask`. To motivate the implementation in Appendix A.2, let us first imagine how to emulate the behaviour of `local`. Our goal is to make the following snippet return the value `15`.

```
let main = handle[hAsk(5)] {
    let x = ask();
    let y = local(double, fn() { ask() });
    add(x, y)
};
```

This means that the `local` operation would need to handle the `ask` effect with the modified value. This is easily achieved, since the innermost handler always applies. If the function to modify the value is called `f`, then the value we should provide to the handler is `f(ask())`.

```
let local = fn(f: fn(Int) Int, g: fn() <Ask|e> a) <|e> a {
    handle[hAsk(f(ask()))] { g() }
}
```

This works but is not implemented as an effect. For example, we cannot modularly provide another implementation of `local`. To turn this implementation into an effect, we start with the effect declaration.

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

It might be surprising that the signature of `local` does not match the signature of the function above. That is because of the call-by-name nature of higher-order operations: instead of a function returning `a`, we simply have a computation that will evaluate to `a`. The effect row is irrelevant and therefore implicit. Now we can provide an elaboration, which is not a function, but better described as a syntactic substitution.

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

Note how similar the elaboration for `local!` is to the `local` function above. In the first line, we specify explicitly what effect the elaboration elaborates (`Reader!`) and which effects should be present in the context where this elaboration is used (`<Ask>`). This can be an effect row of multiple effects if necessary. In this case we only require the `Ask` effect. This means that we can use this elaboration in any expression that is wrapped by at least a handler for `Ask`.

```
let main = handle[hAsk(5)] elab[eLocal] {
    let x = ask();
    let y = local!(double, ask());
    add(x, y)
}
```

That is the full implementation for the higher-order `Reader!` effect in Elaine. Appendix A.2 contains a listing of all these pieces put together in a single example.

Another example is the `Exception!` effect. This effect should allow us to use the `catch!` operation to recover from a `throw`. The latter is an algebraic, so we can start there.

```
type Result[a, b] {
    Ok(a),
    Err(b),
}

effect Throw {
    throw(String) a
}

let hThrow = handler {
    return(x) { Ok(x) }
    throw(s) { Err(s) }
};
```

We assume here that we want to throw some error message, but we could put a different type in there as well. The `throw` operation has a return type `a`, which is impossible to construct in general so it cannot return. The higher-order `Exception!` effect should then look like this:

```
effect Exception! {
    throw!(String) a
    catch!(a, a) a
}
```

In constrast with the `Reader!` effect above, we alias the operation of the underlying algebraic effect here. This makes no functional difference, except that it allows us to write functions with explicit effect rows with `Exception!` and without `Throw`. We might even choose to elaborate to a different effect than `Throw`. The downside is that it requires us to provide the elaboration for the `throw!` operation.

```
let eExcept = elaboration Exception! -> <Throw> {
    throw!(s) { throw(s) }
    catch!(a, b) {
        match handle[hThrow] a {
            Ok(x) => x,
            Err(s) => b,
        }
    }
};
```

We can then use the effect like we used the `Reader!` effect: with an **`elab`** for `Exception!` and a **`handle`** for `Throw`. In the listing below, we ensure that we do not decrement a value of `0` to ensure it will not become negative.

```
let main = handle[hThrow] elab[eExcept] {
    let x = 0;
    catch!({
        if eq(x, 0) {
            throw!("Whoa, x can't be zero!")
        } else {
            sub(x, 1)
        }
    }, 0)
};
```

Since the elaborations can be swapped out, we can also design elaborations with different behaviour. Assume, for instance, that there is a `Log` effect. Then we can create an alternative elaboration that logs the errors it catches, which might be useful for debugging.

```
let eExceptLog = elaboration Exception! -> <Throw,Log> {
    throw!(s) { throw(s) }
    catch!(a, b) {
        match handle[hThrow] a {
            Ok(x) => x,
            Err(s) => {
                log(s);
                b
            }
        }
    }
};
```

We could also disable exception catching entirely if we so desire. This might be helpful if we are debugging a piece of a program that is wrapped in a `catch!` to ensure it never fully crashes, but we want to see errors while we are debugging. Of course, this changes the functionality of the program significantly. We should therefore be careful not to change computations that rely on a specific implementation of the `Exception!`.

```
let eExceptIgnoreCatch = elaboration Exception! -> <Throw> {
    throw!(s) { throw(s) }
    catch!(a, b) { a }
}
```

What these examples illustrate is that elaborations provide a great deal of flexibility, with which we can define and alter the functionality of the `Exception!` effect. We can change it temporarily for debugging purposes or apply another elaboration to a part of a computation. We can also define more `Exception`-like effects and use multiple at the same time.

# 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, add(ask(), 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: $\langle H!_1, \dots, H!_n, A_1, \dots, A_m \rangle$. It then checks the type environment to look for elaborations $E_1, \dots, E_n$ which elaborate $H!_1, \dots, H!_n$, respectively. Only elaborations that are directly in scope are considered, so if an elaboration resides in another module, it needs be imported first. For each higher-order effect, there must be exactly one elaboration.

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

```
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 Appendix B.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

**TODO**: This is currently just an example. The actual transformation needs to be clearly defined too.

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: Untransformed program. This example should use a higher-order effect.

```
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) }
};

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()
    }
};
```

An example of what this transformation is given in listings 6.1 and 6.2, where listing 6.1 shows an untransformed program with higher-order effects and listing 6.2 shows the result of the transformation.

## 6.4 Compiling Elaborations into Handlers

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

While the transformation in the previous section is correct, the transformed program is quite verbose, because the elaboration types need to be passed to every function with higher-order effects. It would be more convenient if this was passed implicitly.

As it turns out, we have a mechanism for passing implicit arguments: algebraic effects! Conceptually, both **`elab`** and **`handle`** are similar: they define a scope in which a given elaboration or handler is used. This scope is the same for both.

To use this observation, we start by defining a handler that returns an elaboration for higher-order effect `A!`, much like the `Ask` effect from Chapter 4.

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, Listing 6.2: Transformed program after compiling elaborations to dictionary passing.

```
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) }
};

# 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 {
            f
        } else {
            let x = f(elab_A);
            fn(elab_A) { x }
        }
    };
    g(elab_A_a)
};
```

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[$e_1$] {$e_2$}
```

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

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

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?

**TODO**: Answer: yes if you care about impredicativity, no otherwise.

# 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?

## 7.1 Monad Transformers

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

In $\lambda$-calculus, effects are not modelled; every function is a function in the mathematical sense, that is, a pure computation (Moggi 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

## 7.2 Monad Transformers

**TODO**: Have to define how operations interact between each pair of transformers: $N^2$ number of implementations for each operation.

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

**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).

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*.

### 7.2.1 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))
```

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

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

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 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: But it's too coarse grained!

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.

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

```
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.

### 7.2.2 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

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 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 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 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.

— (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).

Leijen, Daan (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).

Wadler, Philip (1992). "The essence of functional programming". In: *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '92*. the 19th ACM SIGPLAN-SIGACT symposium. Albuquerque, New Mexico, United States: ACM Press, pp. 1–14. ISBN: 978-0-89791-453-6. DOI: 10.1145/143165.143169.

URL: `http://portal.acm.org/citation.cfm?doid=143165.143169` (visited on 08/24/2023).

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).

# Appendix A

# Elaine Example Programs

This chapter contains longer Elaine samples with some additional explanation.

## A.1 A naive SAT solver

This program is a naive brute-forcing SAT solver. 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 the listing below, we check for solutions of the equation $\neg a \land b$.

```
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());
};
```

## A.2 The Reader effect

```
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()));
};
```

# Appendix B

# 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.

## B.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 B.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.

## B.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{array}{rcl}
\text{program } p & ::= & d \dots d \\
\\
\text{declaration } d & ::= & [\texttt{pub}]\ \texttt{mod}\ x\ \texttt{\{}d \dots d\texttt{\}} \\
& | & [\texttt{pub}]\ \texttt{use}\ x\texttt{;} \\
& | & [\texttt{pub}]\ \texttt{let}\ p\ \texttt{=}\ e\texttt{;} \\
& | & [\texttt{pub}]\ \texttt{effect}\ \phi\ \texttt{\{}s\texttt{,}\ \dots\texttt{,}\ s\texttt{\}} \\
& | & [\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 \\
& | & \texttt{let}\ p\ \texttt{=}\ e\texttt{;}\ es \\
& | & e \\
\text{expression } e & ::= & x \\
& | & \texttt{()} \mid \texttt{true} \mid \texttt{false} \mid \textit{number} \mid \textit{string} \\
& | & \texttt{fn(}p\texttt{,}\ \dots\texttt{,}\ p\texttt{)}\ [T]\ b \\
& | & \texttt{if}\ e\ b\ \texttt{else}\ b \\
& | & e\texttt{(}e\texttt{,}\ \dots\texttt{,}\ e\texttt{)} \mid \phi\texttt{(}e\texttt{,}\ \dots\texttt{,}\ e\texttt{)} \\
& | & \texttt{handler \{return(}x\texttt{)}\ b\texttt{,}\ o\texttt{,}\ \dots\texttt{,}\ o\texttt{\}} \\
& | & \texttt{handle[}e\texttt{]}\ e \\
& | & \texttt{elaboration}\ x\texttt{!}\ \texttt{->}\ \Delta\ \texttt{\{}o\texttt{,}\ \dots\texttt{,}\ o\texttt{\}} \\
& | & \texttt{elab[}e\texttt{]}\ e \mid \texttt{elab}\ e \\
& | & 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 \\
& | & \texttt{()} \mid \texttt{Bool} \mid \texttt{Int} \mid \texttt{String} \\
& | & \texttt{fn(}T\texttt{,}\ \dots\texttt{,}\ T\texttt{)}\ T \\
& | & \texttt{handler}\ x\ \tau\ \tau \\
& | & \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{array}
$$

Figure B.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:

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

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 $\langle A_1, \ldots, A_n | e_A\rangle$ is represented as the multiset $\{A_1, \ldots, A_n\} + e_A$.

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 $A$.

## B.3 Typing judgments

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

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

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

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

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

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.

### B.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.

### B.3.2 Expressions

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

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

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

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

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

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

### B.3.3 Declarations and Modules

The modules are gathered into $\Gamma_M$ and the variables that are in scope are gathered in $\Gamma_V$. Each module has 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.

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

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

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

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

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

### B.3.4 Algebraic Effects and Handlers

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

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

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

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

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

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

### B.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 $\Delta$ they elaborate into. The operations do not get a continuation, so the type checking is a bit different there. As arguments, they take the effectless types they specified along with the effect row $\Delta$. Elaborations are not added to the value context, but to a special elaboration context mapping the effect identifier to the row of effects to elaborate into.

**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.

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

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

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

## B.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:

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

## B.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

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

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

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

## B.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 B.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 B.2: Overview of the functions in the `std` module in Elaine.