

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



Terts Diepraam
September 18, 2023

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

© 2023 Terts Diepraam.

Cover picture: Rubin's Vase.

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. M. Spaan, Faculty EEMCS, TU Delft
Committee Member: Dr. C. Bach Poulsen, Faculty EEMCS, TU Delft
University Supervisor: Ir. C. van der Rest, Faculty EEMCS, TU Delft

Contents

Contents	iii
1 Introduction	1
2 Algebraic Effects	7
2.1 Monads	7
2.2 Effect Composition with the Free Monad	11
2.3 Algebraic Effects	14
2.4 Building a Language with Algebraic Effects	15
3 Higher-Order Effects	19
3.1 Computation Parameters	19
3.2 Breaking Algebraicity	20
3.3 The Modularity Problem	21
3.4 Hefty Algebras	21
4 A Tour of Elaine	25
4.1 Overview	25
4.2 Basics	25
4.3 Types	28
4.4 Algebraic Data Types	28
4.5 Recursion & Loops	29
4.6 Algebraic Effects	30
4.7 Functions Generic over Effects	33
4.8 Higher-Order Effects	34
5 Implicit Elaboration Resolution	39
6 Related Work	41
6.1 Monad Transformers	41
6.2 Other Solutions to the Modularity Problem	42
6.3 Languages with First-Class Effects	43
6.4 Effects as Free Monads	44
7 Conclusion	45
Bibliography	47
A Elaine Example Programs	51

A.1	A naive SAT solver	51
A.2	The Reader Effect	52
A.3	The Writer Effect	53
A.4	Structured Logging	53
A.5	Parser Combinators	54
B	Elaine Specification	57
B.1	Syntax definition	57
B.2	Effect row semantics	57
B.3	Typing judgments	60
B.4	Desugaring	63
B.5	Semantics	63
B.6	Standard Library	64

Chapter 1

Introduction

In many programming languages, computations are allowed to have *effects*. This means that they can perform operations besides producing output, and interact with their environment. A computation might, for instance, read or modify a global variable, write to a file, throw an exception, or even exit the program.

Historically, programming languages have supported effects in 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), who argues 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: 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 a program on the programmer. If the language cannot provide any guarantees about what (a part of) a program can do, the programmer has to check instead. For instance, if a function somewhere in the code sets global variable to some incorrect value. This can then cause seemingly unrelated parts of the program to 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. In languages where this is possible, effectful operations therefore limit our ability to split the code into chunks to be examined separately.

A solution is to treat effects in a more structured manner. For example, instead of allowing `goto`, a language might provide exceptions. In a language like Java, checked exceptions are part of the type system, so that the type checker can verify that all exceptions are handled. However, with this approach, any effect must be backed by the language. That is, the language needs to have a dedicated feature for every effect that should be supported in this way and new effects cannot be created without adding a new feature to the language. This means that the support for various effects is limited to what the language designers have decided to add.

In contrast, languages adhering to the functional programming paradigm disallow effectful operations altogether.¹ 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*, meaning that it always returns identical outputs for identical inputs and does not interact with the environment. By requiring that all functions are pure, a type signature of a function becomes almost a full specification of what the function can do.

However, sometimes effectful operations are still desired. Consider the following program written in Koka, a functional language where function need to be pure. In this program,

¹Usually there are some escape hatches to this rule, such as Haskell’s `trace` function, which is built-in and effectful, but only supposed to be used for debugging.

there is a set of users that are considered administrators. The `all_admins` function checks whether all user ID's in a list are administrators.

```
1  val admins = [0,1,2]
2
3  fun is_admin(user_id: int): bool
4    admins.any(fn(x) x == user_id)
5
6  fun all_admins(l: list<int>): bool
7    l.map(is_admin).foldl(True, (&&))
8
9  val result = all_admins([0,1,2,3])
```

Koka

This is a fairly standard functional program where the result is a single boolean. However, the program does not tell us which users are not admins, which could be useful information to print. In an imperative language, we could just add a `print` call in `is_admin` to log any user that was not an admin and call it a day. But in a functional language, we cannot do this. Instead, each message we want to log needs to be returned by `is_admin`. These messages then need to be concatenated to build up the string that should be printed.

```
1  fun is_admin(user_id: int): (bool, string)
2    if admins.any(fn(x) x == user_id)
3    then (True, "")
4    else (False, "Denied " ++ show(user_id) ++ "\n")
5
6  fun all_admins(list: list<int>): (bool, string)
7    match list
8    Nil() -> (True, "")
9    Cons(x, xs) ->
10      val (y, s) = is_admin(x)
11      val (ys, s') = all_admins(xs)
12      (y && ys, s ++ s')
13
14  val (result, log) = all_admins([1,2,3,4])
```

Koka

So, adding some logging made the program much more complicated. For larger programs, one might imagine that programming with effects in a functional language therefore quickly becomes laborious. Additionally, the functions above are adapted specifically to our logging effect; using any other effect would require a different implementation. Therefore, we should abstract over the effects in the computation.

This abstraction can be found in the form of *monads* (Peyton Jones and Wadler 1993; Wadler 1992). A monad represents a computation with some effect. It is a type constructor that takes the return type of the computation as a parameter. For a type to be a monad, it needs to define two functions: `return` and `>>=`. The former wraps a value in the monad and the latter sequences 2 monadic computations. In Koka, we cannot call these functions `return` and `>>=`, so we call them `pure` and `bind`, respectively.

```

1  alias log<a> = (v: a, msg: string)
2
3  fun pure(v: a): log<a>
4    (v, "")
5
6  fun bind(m: log<a>, k: a -> log<b>): log<b>
7    val (a, s) = m
8    val (b, s') = k(a)
9    (b, s ++ s')
10
11 fun log(msg: string): log<()>
12  ((), msg)

```

Koka

60

61 The `is_admin` and `all_admins` can then be written using these functions instead of dealing
62 with the strings in the tuples directly. Hence, we have abstracted over the effect and could
63 replace it with another. Specifically, we could change the effect that `is_admin` uses without
64 changing `all_admins`.

```

1  fun all_admins(list)
2    match list
3      Nil() -> pure(True)
4      Cons(x, xs) -> is_admin(x).bind fn(y)
5        all_admins(xs).bind fn(ys)
6        pure(y && ys)
7
8  fun is_admin(user_id: int): log<bool>
9    if admins.any(fn(x) x == user_id)
10   then pure(True)
11   else
12     log("Denied " ++ show(user_id) ++ "\n").bind fn(())
13     pure(False)

```

Koka

65

66 In fairness, Koka is not built for monadic operations and other languages provide more
67 convenient syntax for monads. However, the structure of the same program in such a language
68 would be roughly the same.

69 Another limitation of the monad approach becomes apparent when we want to use multi-
70 ple effects. The problem is that the composition of two monads does not yield a monad. This
71 is a limitation that can be worked around with *monad transformers*. A monad transformer
72 takes a monad and adds operations to it. The operations of every effect then need to be
73 implemented on every transformer. Adding a single effect therefore requires additional imple-
74 mentations of its operations every other monad transformer. The number of implementations
75 therefore grows quadratically with the number of effects.

76 To overcome these limitations, we instead turn to the theory of *algebraic effects*, which
77 allow effects to be defined modularly. In this theory, an effect consists of a set of *effect*
78 *operations*. A computation using an effect then needs to be wrapped in a *handler*, which
79 defines the semantics for the operations. These modular effects and handlers are based on
80 the free monad. It is possible to encode the free monad in Haskell and use algebraic effects
81 in Haskell that way.

82 However, Koka supports algebraic effects as a first-class construct. This allows Koka to
83 make the use of algebraic effects very easy. It is not the only language to do this; other
84 examples include Frank (Lindley, McBride, and McLaughlin 2017), Effekt (Bach Poulsen

and van der Rest 2023), Eff (Bauer and Pretnar 2015), Helium (Biernacki et al. 2019), and OCaml (Sivaramakrishnan et al. 2021). In the listing below, we first declare the algebraic effect `log`. This effect has a single operation also called `log`, which takes the message to log as an argument. Then we define a handler `hLog` for the `log` effect. The handler transforms the effectful computation into a monad, which matches our tuple from before. Note that the `return` branch matches the `pure` function and that the `log` branch combines the `bind` and `log` functions in our monad implementation. The `is_admin` and `all_admin` functions then simply declare that they use the `log` effect, which allows them to use the `log` operation. This relies on the fact that Koka’s `map` and `foldl` functions are generic over effects in the computation.

```

1  effect log
2    ctl log(msg: string): ()
3
4  val hLog = handler
5    return(x) (x, "")
6    ctl log(msg)
7      val (x, msg') = resume(())
8      (x, msg ++ msg')
9
10 fun is_admin(user_id: int): <log> bool
11   val result = admins.any(fn(x) x == user_id)
12   if !result then
13     log("Denied " ++ show(user_id) ++ "\n")
14   result
15
16 fun all_admins(l): <log> bool
17   l.map(is_admin).foldl(True, (&&))
18
19 val (result, log) = hLog { [1,2,3,4].all(is_admin) }
```

Koka

In the end, the implementation then looks very much like imperative code, but the type system still resembles the type system of functional languages. Effects are handled in a structured way, but are still convenient to use. There are several other advantages too. The effects are modular and can be combined easily. Additionally, the handlers are modular; any handler can be swapped out for another handler, changing the semantics of the effect. For example, we could write a handler that ignores all `log` calls or stores the logged messages in a list.

However, some effects are not algebraic and can therefore not be represented as effects in a language like Koka. *Higher-order effects* are effects with operations that take effectful computations as arguments, and they are not algebraic in general. As Plotkin and Pretnar (2009) have shown, they can be written as handlers, but not as effect operations. This is known as the *modularity problem* for higher-order effects (Wu, Schrijvers, and Hinze 2014). Several extensions to algebraic effects have been proposed to accommodate 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. Elaborations give semantics to higher-order effects by translating them into computations with only algebraic effects. This means that evaluation of a computation becomes a two-step process: first higher-order effects are elaborated into algebraic effects, which can then be handled. Like handlers, elaborations are modular, and it is possible to define multiple elaborations for a single effect.

Therefore, there currently exist languages with algebraic effects and there is a theory for

117 hefty algebras, but there is no language yet based on hefty algebras. This is the gap in the
118 research that this thesis aims to fill. The question we therefore wish to answer is:

119 **How can we design a language with higher-order effects and elaborations with**
120 **hefty algebras as underlying theory?**

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

natural?
hmm

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

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

137 We present a specification for Elaine, including the syntax definition, typing judgments,
138 and semantics. Along with this specification, we provide a reference implementation written
139 in Haskell in the artefact accompanying this thesis. This implementation includes a parser,
140 type checker, interpreter, pretty printer, and the transformations mentioned above. Elaine
141 opens up exploration for programming languages with higher-order effects. While not a viable
142 general purpose language in its own right, it can serve as inspiration for future languages.

FEEDBACK:
Casper: clar-
ify why not,
omit, or re-
place by a
statement
that it is a
prototype-
/intended
as a demon-
stration/ve-
hicle for re-
search/... if
that is what
you mean.

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

- 145 • We define a syntax suitable for a language with both handlers and elaboration (Ap-
146 pendix B.1).
- 147 • We provide a set of examples for programming with higher-order effect operations.
- 148 • We present a type system for a language with higher-order effects and elaborations,
149 based on Hindley-Milner type inference and inspired by Koka's type system. This
150 type system introduces a novel representation of effect rows as multiset which, though
151 semantically equivalent to earlier representations, allows for a simple definition of effect
152 row unification.
- 153 • We propose that elaborations should be inferred in most cases and provide a type-
154 directed procedure for this inference (Chapter 5).

FEEDBACK:
Casper: Ref-
erence chap-
ters for every-
thing.

FEEDBACK:
Casper:
Sharpen ex-
planation of
multi-set se-
mantics.

155 This thesis consists of the following parts. First, we give an overview of the relevant theory
156 of algebraic effects in Chapter 2 and higher-order effects and hefty algebras in Chapter 3.
157 Then, we present Elaine in Chapter 4. The implicit elaboration resolution is then discussed
158 in Chapter 5. Finally, we discuss related work in Chapter 6 and conclude in Chapter 7.
159 The appendices contain additional examples of Elaine programs (Appendix A) and a full
160 specification of the language (Appendix B).

Artefact The artefact accompanying this thesis contains a full prototype implementation for Elaine, written in Haskell. The `README.md` file contains instructions for building and executing the interpreter.

The source code of the parser, type checker, interpreter, and other aspects of the implementation can be found in the `src/Elaine` directory. The `examples` directory contains various example programs written in Elaine, including implementations of the reader effect, writer effect, exception effect, structured logging, and a set of parser combinators.

The artefact is available online at <https://github.com/tertsdiepraam/elaine>.

Chapter 2

Algebraic Effects

The theory of algebraic effects is intended to make working with effectful operations easier by making effects composable. It achieves this goal for many important effects, but, crucially, does not cover higher-order effects; effect operations that take other effectful computations as arguments. The theory of hefty algebras extends the theory of algebraic effects with higher-order effects. How this is achieved is discussed in Chapter 3. Since Elaine is based on the theory of hefty algebras, the theory of algebraic effects also applies to Elaine. In this chapter, we give an introduction to algebraic effects. In the next chapter, we discuss its limitations regarding higher-order effects and describe how hefty algebras overcome those limitations.

FEEDBACK:
Lucas: Say that it requires some familiarity with functional programming syntax at least

2.1 Monads

TODO: Some more background and citations on monads

TODO: Who are we basing this on?

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

While many descriptions of monads using category theory and various analogies can be employed in explaining them, for our purposes, a monad is a type constructor `m` with two associated functions: **return** and `>>=`, with the latter pronounced “bind”. In Haskell, this concept is easily encoded in a type class, which is listed below.

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

Haskell

This type class tells us that we can construct a value of `m a` for any type `a` and for any monad `m` using **return**. This represents a computation that has no further effects and just “returns” a value. Additionally, we can compose two monadic computations using `>>=`, which takes a monadic computation and a *continuation*, which is the function that should be called after the operation has been performed. The continuation is passed the return value of the operation as an argument. The `>>=` operator therefore sequences two monadic operations.

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 and returns immediately once the **abort** operation is used. To represent the intention of the operation, we define the **abort** operations as **Nothing**.

```

1  data Maybe a
2    = Just a
3    | Nothing
4
5  class Monad Maybe where
6    return = Just
7
8    Just a  >>= k = k a
9    Nothing >>= k = Nothing
10
11 abort :: Maybe a
12 abort = Nothing

```

Haskell

With this definition, we can chain functions returning **Maybe**. For example, we can define a **head** function with the type `[a] -> Maybe a` 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 `>>=`.

```

1  head :: [a] -> Maybe a
2  head (x:xs) = Just x
3  head _ = Nothing
4
5  safeDiv :: Int -> Int -> Maybe Int
6  safeDiv _ 0 = Nothing
7  safeDiv x y = Just $ div x y
8
9  main = do
10    print $ head [] >>= safeDiv 10 -- -> Nothing
11    print $ head [0,1,2] >>= safeDiv 10 -- -> Nothing
12    print $ head [2,3,4] >>= safeDiv 10 -- -> Just 5

```

Haskell

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 of type `s` as an argument and return a new value 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)`. Instead of modifying the state directly, it maps an old state to a new 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.

```

1  -- Increment the state by `a` and return the old state
2  inc :: Int -> Int -> (Int, Int)
3  inc a s = (s + a, s)
4
5  multipleIncs :: Int -> (Int, Int)
6  multipleIncs s = let
7    (s', _) = inc 5 s
8    (s'', _) = inc 6 s'
9    in inc 7 s''

```

Haskell

The program becomes verbose and repetitive as a result. However, all functions types that use state now end with the same pattern: `s -> (s, b)`. This is an opportunity for abstraction,

216 because we can define a type for that pattern. Since this type represents the state effect, this
 217 type is called `State`.

```
1 newtype State s a = State (s -> (s, a))
2
3 -- so that the inc function becomes:
4 inc :: Int -> State Int Int
5 inc a = State (\s -> (s + a, s))
```

Haskell

218

219 Now we can turn `State s` into a monad! We do this by making it an instance for the **Monad**
 220 type class and implementing the **return** and `>>=` functions for it. This allows us to compose
 221 functions returning the `State` type. Additionally, we define the **get** and **put** operations,
 222 which are the basic building blocks we can use to build more complex computations.

```
1 instance Monad (State s) where
2   return x = State (\s -> (s, x))
3
4   State fa >>= k = State $ \s ->
5     let (s', a) = fa s
6         State fb = k a
7     in fb s'
8
9   get :: State s s
10  get = State (\s -> (s, s))
11
12  put :: s -> State s ()
13  put s = State (\_ -> (s, ()))
```

Haskell

223

224 To retrieve the final value of the computation, we define a function `runState`, which takes
 225 an initial state and returns a pair of the final state and the returned value.

```
1 runState :: s -> State s a -> (s, a)
2 runState initialState (State func) = func initialState
```

Haskell

226

227 The `inc` operations can then be sequenced using the `>>=` operator. Because the return value
 228 of `inc` is irrelevant in the computation, we define a shorthand operator `>>`, which ignores the
 229 return value of the first operation.

```
1 (>>) :: Monad m => m a -> m b -> m b
2 a >> b = a >>= \_ -> b
3
4 inc :: Int -> State Int Int
5 inc x = get >>= \s -> put (s + x) >>= return s
6
7 multipleIncs :: State Int Int
8 multipleIncs = inc 5 >> inc 6 >> inc 7
9
10 main = print (runState 0 bar) -- prints 0 + 5 + 6 + 7 = 18
```

Haskell

230

231 This is the power of monads: they allow us to abstract the effectful operations away, while
 232 also signalling the effects that a function requires in the return type. In the final example, we
 233 do not have to think about how the `State` monad works any more, but only use the **get** and

`put` operations to build complex computations. The abstraction separates the interface from the implementation. This modularity is one of the core motivations of the study of effects.

To make working with monads more convenient, Haskell also features `do`-notation, which is syntactic sugar for the `>>=` and `>>` operators. Using `do`-notation, the `multipleIncs` computation from the previous example can be written as:

```
1 multipleIncs = do
2   inc 5
3   inc 6
4   inc 7
```

Haskell

If the results from the `inc` computations needs to be used, the `<-` operator, which is part of `do`-notation, can be used to bind the result of a computation to a variable. For example, the sum of all the results from the `inc` calls can be returned.

```
1 multipleIncs = do
2   a <- inc 5
3   b <- inc 6
4   c <- inc 7
5   return (a + b + c)
6
7 -- which is equivalent to
8 multipleIncs =
9   inc 5 >>= \a ->
10    inc 6 >>= \b ->
11    inc 7 >>= \c ->
12    return (a + b + c)
```

Haskell

This is a convenient method for programming with effects in Haskell, while also staying true to its functional paradigm. However, monads are also limited, since they cannot be composed. Imagine, for instance, a computation that decrements some state and returns the new value, but also asserts that the value never becomes negative and returns **Nothing** in that case. This computation might look as follows.

```
1 decrement :: State Int (Maybe Int)
2 decrement = get >>= \s ->
3   if s > 0
4   then put (s - 1) >> return (return (s - 1))
5   else return abort
```

Haskell

What is important here is that **Maybe** cannot benefit from the `>>=` operator. The type of `decrement` is not a combined monad for both effects, but one monad wrapped in another. For complex computations, this quickly gets complicated. Instead, there could be some combined monad `MaybeAndState` that combines the operations of both monads.

```
1 decrement :: MaybeAndState Int Int
2 decrement = get >>= \s ->
3   if s > 0
4   then put (s - 1) >> return (s - 1)
5   else abort
```

Haskell

While it is technically possible to define such a monad, we would need to define one for every combination of monad operations that we would like to use, which quickly becomes very cumbersome. Hence, we need to look elsewhere for a solution. One solution to this is to use monad transformers, as explained in Section 6.1. Another solution is to use the *free monad*.

2.2 Effect Composition with the Free Monad

TODO: Cite Casper's blog or a more academic version of it from somewhere. <http://casperbp.net/posts/2023-07-algebraic-effects/index.html>

TODO: Give this description another go

The free monad is a monad that encodes the structure of a program without imposing semantics ^{citation needed}. The free monad takes a functor `f` as an argument. The free monad then gives a syntactic description of the operations given by that functor. It is therefore the trivial monad parametrized by the operations of `f`. In Haskell, the free monad is implemented as the `Free` data type.

```
1 data Free f a
2   = Pure a
3   | Do (f (Free f a))
4
5 instance Functor f => Monad (Free f) where
6   return = Pure
7   Pure x >>= f = f x
8   Do g >>= f = Do (fmap (>>= f) g)
```

Haskell

Given some `State s` functor, then `Free (State s)` is a monad. Of course, this is only useful if the `State s` functor can generate a monad with the same functionality as the original state monad. To do so, we define a data type with the two constructors of `State`. This is a functor over the `k` parameter, which represents the *continuation* of the computation, which is the rest of the computation to be evaluated after the effect operation. Note that we do not have to give definitions of `return` and `>>=` since those are defined generically for any `f` on `Free`. We only have to derive the default `Functor` instance.

```
1 data State s k = Put s k | Get (s -> k)
2   deriving Functor
```

Haskell

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`. The `Abort` constructor does not use the continuation because it signals that evaluation should stop.

```
1 data Abort k = Abort
2   deriving Functor
```

Haskell

In contrast with monads, these functors can be meaningfully composed. We define a type-level operator `+`, which represents a coproduct for functors. This operator can be thought of as `Either` for functors, since `Either` is the coproduct for types. We use this operator to build lists of functors. Just like lists have a `Cons` and `Nil`, these lists consist of `+` and `End`, where `End` is a functor without any constructors.

FEEDBACK:
We only need constructors for operations, hence only one constructor for one operation

```

1  infixr 6 +
2  data (f + g) a = L (f a) | R (g a)
3      deriving Functor
4
5  data End k
6      deriving Functor

```

Haskell

The `End` functor has the property that it does not add any operations. Therefore, we have that `Free (f + End)` is functionally the same as `Free f` and that `Free End a` is equivalent to `a`. We can then make monads for any combination of the functors we have defined, such as `Free (State s + End)`, `Free (Abort + End)` or `Free (State s + Abort + End)`. In general, we can construct a monad for any set of functors.

However, we have no way to use any of the effect operations for this functor. For example, if we have `Free (State s + Abort + End)`, how would we use the `get` operation that we expect from the state monad? 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 type class 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`. These convenience functions are called *smart constructors*.

TODO: Parts of this might not be important

```

1  class f < g where
2      inj :: f k -> g k
3
4  instance f < f where inj = id
5  instance f < (f + g) where inj = L
6  instance f < h => f < (g + h) where inj = R . inj
7
8  get :: State s < f => Free f s
9  get = Do $ inj $ Get Pure
10
11 put  :: State s < f => s -> Free f ()
12 put s = Do $ inj $ Put $ Pure ()
13
14 abort :: Abort < f => Free f ()
15 abort = Do $ inj $ Abort

```

Haskell

This makes it possible to construct a computation using all those operations. For example, a computation that checks the state, asserts that it is larger than 0, and then decrements the state by 1.

```

1  decrement :: Free (State Int + Abort + End) Int
2  decrement = get >>= \s ->
3      if s > 0
4      then put (s - 1) >>= return (s - 1)
5      else abort

```

Haskell

However, there is no way to evaluate this computation, because the free monad is just a syntactic representation of the computation. Hence, we need to define an algebra for the `Free` data type. To do that, there needs to be a function with the type

$$\text{Free } (f + f') \text{ } a \rightarrow \text{Free } f' \text{ } b$$

for every `f` and finally a function `Free End a -> a` to reduce the free monad to a final value. Following Plotkin and Pretnar (2009), these functions are called *handlers*. In general, any handler operating on `Free` needs to take two cases into account, since `Free` has two constructors: `Pure` and `Do`. By using a fold over `Free` we can define a handler in terms of two smaller functions:

- the return case, `a -> Free f' b`;

- and the case for handling the operations: `f (Free f' b) -> Free f' b`.

However, to generalize the handler, we add a parameter `p` as well, which is a parameter that the handlers can access and modify. This parameter is used to thread state through the computation. Therefore, `handle` is defined as follows.

```

1 fold :: Functor f => (a -> b) -> (f b -> b) -> Free f a -> b
2 fold gen _ (Pure x) = gen x
3 fold gen alg (Op f)  = alg (fmap (fold gen alg) f)
4
5 handle :: (Functor f, Functor f')
6         -- a function for the return case:
7         => (a -> p -> Free f' b)
8         -- a function for handling operations:
9         -> (f (p -> Free f' b) -> p -> Free f' b)
10        -- the type of the resulting handler:
11        -> Free (f + f') a -> p -> Free f' b
12 handle ret f = fold ret $
13   \case
14     L x -> f x
15     R x -> \p -> Do $ fmap (\m -> m p) x

```

Handlers for the various effects can then be constructed using `handle`. For each computation, we need a handler for each effect to remove them from the free monad so that only the `End` functor remains. Then we can reduce `Free End a` to `a`. So, the `decrement` function above requires handlers for `State s` and `Abort`.

```

1  -- The Do case does not need to be handled since End
2  -- cannot be constructed
3  handleEnd :: Free End a -> a
4  handleEnd (Pure a) = a
5
6  handleAbort :: Functor f => Free (Abort + f) a -> Free f (Maybe a)
7  handleAbort c = handle
8    (\a _ -> Pure $ Just a)
9    (\Abort () -> Pure Nothing)
10   c ()
11
12  handleState :: Functor f => s -> Free (State s + f) a -> Free f (s, a)
13  handleState = flip $ handle
14    (\x s -> pure (s, x))
15    (\x s -> case x of
16      Put s' k -> k s'
17      Get k -> k s s)
18
19  result :: (Int, Maybe Int)
20  result = handleEnd $ handleState (0::Int) $ handleAbort decrement

```

Haskell

323

324 This finally allows us to use the abort and state effects together, while providing a handler
 325 per effect. Note that the order in which the handlers are applied matters for the return type
 326 of the result. If abort is handled first and state second, the final type is **(Int, Maybe Int)**.
 327 If state is handled first, it is **Maybe (Int, Int)**.

328 While the plumbing needed for a free monad is extensive, it has many advantages over
 329 regular monads. First, we can combine multiple functors in our type signatures. Second, we
 330 can define operations that work for any effect composition that contains an effect. Third, we
 331 can provide modular handlers that handle a single effect from the composed functors. If all
 332 effects are defined in this way, then effect is automatically compatible with all other effects.

333 Finally, we have not only gained modularity for the effects themselves, but also for the
 334 handlers. The effects have become an interface, while the handlers provide the semantics.
 335 Within this framework, the semantics of effects can be changed without touching the type
 336 and definition of the computation. There is nothing preventing different implementations of
 337 the handlers. It is, for example, possible to define a state handler in which put operations
 338 are ignored, keeping the state is constant.

339 2.3 Algebraic Effects

340 The free monad encoding in the previous section is an implementation of algebraic effects in
 341 Haskell. The term “algebraic” comes from the fact that this method works for effects that
 342 can be described as algebraic theories (Plotkin and Power 2001). Later, Plotkin and Power
 343 (2003) showed that this is only possible for effects that satisfy the *algebraicity property*.

344 The algebraicity property states that the $\gg=$ operation distributes over the computation
 345 parameters of an operation. This means that if there is some operation **op** that has some pa-
 346 rameter of type **k** then the following computations should be equivalent for some continuation
 347 **k'**:

$$(\text{op } k) \gg= k' \quad == \quad \text{op } (k \gg= k')$$

348 So, if the state effect satisfies the algebraicity property, the following equality should hold for

any continuations k and k' :

$$(\text{Do } (\text{Put } s \ k)) \gg= k' \quad == \quad \text{Do } (\text{Put } s \ (k \gg= k'))$$

Intuitively, this matches how we expect state to behave: if the state is changed, it will remain changed throughout the rest of the continuation, until it is changed again.

By construction, the algebraicity property holds for any effect we have defined in the previous section. This can be derived from the definitions of $\gg=$ on `Free` and `fmap` for the effects. Indeed, we can apply the definitions to `Free (State s)` to verify that the algebraicity property holds.

```

1 | Do (Put s k) >>= k'
2 | -- apply >>= of Free:
3 | Do (fmap (>>= k) (Put s k'))
4 | -- apply fmap of Put:
5 | Do (Put s (k >>= k'))

```

Haskell

Consequently, any effect that does not satisfy the algebraicity property cannot be written as the free monad. While the state and abort effects satisfy this property, *higher-order effects*, do not. Examples of higher-order effects are exception catching and the reader effect with a local operation. Those effects are discussed in Chapter 3.

2.4 Building a Language with Algebraic Effects

Although the previous sections contain an encoding of algebraic effects, there are details in this encoding that we might like to hide. For instance, writing all return types as `Free f a` for every function becomes repetitive. Every returned value also needs to be wrapped in `pure` to be mapped to the monad. Our goal is then to remove as much of the additional syntax that is required to work with effects when compared to pure functions.

This is where we reach the limits of what we can achieve with the encoding of the free monad in Haskell. If we instead design a new language which integrates algebraic effects as a core feature in the language, we have much more freedom in designing a syntax and type system that work well for this purpose.

Elaine is a language with support for algebraic effects, but it also supports higher-order effects. Therefore, this section focuses on Koka (Leijen 2014, 2023), which only supports algebraic effects. Since Elaine is heavily inspired by Koka, the same concepts apply to Elaine.

At the core of such languages lies the following concept: all functions implicitly return the free monad with some effects. Therefore, we write $a \rightarrow e \ b$, which is analogous to $a \rightarrow \text{Free } e \ b$ in the Haskell encoding. In that signature, we call e the *effect row* and its elements as *effects*. So, the function signature $a \rightarrow e \ b$ 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 Koka, we will write effect rows as

$$\langle e_1, e_2, \dots, e_n \rangle.$$

In the type system, we are then allowed to use different orders of effects interchangeably. This is a clear ergonomic improvement over the free monad encoding, where we could only reason about inclusion of one effect at a time.

In such a language, effects are a special construct separate from monads and functors. Therefore, effect rows can get special treatment in the type system. It should be able to, for example, reason about equality between effect rows with the same effects in different orders, such as $\langle a, b \rangle$ and $\langle b, a \rangle$.

All the effects in this row are single effects, they are not composed. In Haskell, this is not the case, some functor f can represent a composed functor. Therefore, we need notation to express that an effect row can be extended with another effect row. This is written as

$$\langle e_1, e_2, \dots, e_n | es \rangle,$$

where es is the tail of the effect row; a variable representing the effect row with which this effect row can be extended.

We can define the same effects as before, like state and abort, but in Koka, we do not define them as functors. Instead, we define them using the **effect** keyword and each constructor of the functors is then declared with the **ctl** keyword.

```

1  effect abort
2    ctl abort(): a
3
4  effect state<a>
5    ctl get(): a
6    ctl put(x: a): ()

```

Koka

The equivalent of `Free (State s + Abort + End) a` is then $\langle \text{state}\langle s \rangle, \text{abort} \rangle a$. The equivalent of a handler would then be a function which takes $() \rightarrow \langle f | e \rangle a$ and returns $\langle | e \rangle a$. In Koka, such a function can be defined with the **handler** construct, which requires an implementation for each operation of an effect and a special function for the return case.

Note the similarity to the `handle` function we defined in Haskell before. In the case of abort effect, this handler is assigned to variable for later use. The state handler resembles the original state monad: it takes a computation $() \rightarrow \langle \text{state}\langle s \rangle | e \rangle a$ and returns $s \rightarrow e(s, a)$. For example, the return arm yields the anonymous function $\text{fn}(x)(s, x)$. The continuation can be called in a handler with the `resume` function in Koka. Since handling is defined by a fold, just like in the Haskell encoding, the effect is already handled in the continuation. For the state effect, `resume` therefore returns a function with the type $s \rightarrow e(s, a)$.

```

1  val hAbort = handler
2    return(x)  Just(x)
3    ctl abort() Nothing
4
5  fun hState(init, c)
6    fun h(c')
7      with handler
8        return(x)  fn(s) (s, x)
9        ctl get()  fn(s) resume(s)(s)
10       ctl put(n) fn(_) resume(())(n)
11       c'()
12
13  h(c)(init)

```

Koka

In the free monad encoding in Haskell, the state had to be passed through the handlers as a parameter. Koka is a bit more flexible and allows us to return values with effectful computations. Therefore, it does not need the additional parameter.

Koka helps us hide some details, but the structure in the listing above is largely the same as with the free monad encoding. The larger differences become apparent when we want to use the effects in some computation. A port of the decrement function is listed below.

```

1 fun decrement(): <state<int>,abort> int
2   val s = get()
3   if s == 0 then
4     abort()
5
6   put(s - 1)
7   s - 1
8
9 val x = hAbort { hState(3, decrement) } // -> Just(2)
10 val y = hAbort { hState(0, decrement) } // -> Nothing

```

418

419 The `>>=` operator is entirely implicit here. Therefore, it is similar to Haskell's `do`-notation.
 420 However, in `do`-notation, every effectful operation needs to be on a separate line. For example,
 421 if the state needs to be incremented by 1, this can be achieved in one line in Koka, but in
 422 Haskell using `do`-notation requires two lines.

```

1 | put(get() + 1)

```

423

```

1 do
2   x <- get
3   put (x + 1)

```

424

425 In Koka, effectful operations can be used anywhere as long as they are wrapped in a corre-
 426 sponding handler. In the end, the syntax is closer to imperative programming languages than
 427 functional programming languages. However, the type system still very much resembles that
 428 of a functional language. This is important because this means that we have not lost any of
 429 the type safety that the monadic treatment of effects provides. The signature of a function
 430 in Koka still gives a complete specification of all effects that a function might perform. In
 431 imperative languages, this information is entirely missing from the function signature. For
 432 example, the type system can assert that a function is entirely pure. In the listing below,
 433 the `<>` in the type of the function asserts that it does not require effects, yet the `println`
 434 function requires an effect. Hence, Koka's type checker will yield a type error.

```

1 fun should_be_pure(x: int): <> int
2   println("This will give a type error!")
3   x + 10

```

435

436 As will become clear in Chapter 4, Elaine takes a lot of inspiration from Koka. Handlers and
 437 effects are defined in the same way, modulo some syntactical difference. What sets Elaine
 438 apart, is that it also supports higher-order effects, which will be explained in the next chapter.

Chapter 3

Higher-Order Effects

In the previous chapter, we explained the concept of algebraic effects; effects that satisfy the algebraicity property. We also mentioned that not all effects are algebraic. To be more specific, the effects that are not algebraic are higher-order effects: effects that take effectful computations as parameters. As a result, it is not possible to give modular implementations for these operations, like we can with algebraic effects. This chapter details the difficulties around higher-order effects and discusses hefty algebras, the theory that Elaine is based on.

3.1 Computation Parameters

Recall that an effect in the free monad encoding is a functor over some `k` with some constructors. The type `k` represents the continuation of the computation. Naturally, it is possible to write a constructor with multiple parameters of type `k`. For example, we could make a `Branch` functor which takes a boolean and two computations. Based on the boolean value, it selects the branch to evaluate. It is essentially an **if-else** expression expressed as an effect.

```
1 data Branch k = Branch Bool k k
2
3 branch :: Branch < f => Bool -> Free f a -> Free f a -> Free f a
4 branch b ifTrue ifFalse :: Do $ inj $ Branch b ifTrue ifFalse
```

Haskell

The important observation with this effect is that both `ifTrue` and `ifFalse` behave like continuations. To examine why, consider the following computation.

```
1 branch b (pure 0) (pure 1) >>= \x -> pure (x + 1)
```

Haskell

Like previously established, the `>>=` operator distributes over the computation parameters. This yields the following expression.

```
1 branch b
2   (pure 0 >>= \x -> pure (x + 1))
3   (pure 1 >>= \x -> pure (x + 1))
4 -- which reduces to
5 branch b (pure 1) (pure 2)
```

Haskell

This computation has the same intended semantics as the original. The distribution of `>>=` therefore does not change the semantics and hence the effect is algebraic. Therefore, there would be no problem encoding this effect in Haskell using the encoding in the previous chapter and, by extension, in Koka.

This is what we mean by saying that the parameters are computation-like: the continuation can be appended to it without changing the semantics of the effect.

3.2 Breaking Algebraicity

For other effects, however, the intended semantics are not such that the computation parameters are continuation-like. These effects are called higher-order effects [citation needed](#).

One such effect is the **Reader** effect. Traditionally, the **Reader** monad has two operations: **local** and **ask**. The latter functions much like the **get** operation from the state effect and is algebraic. However, the **local** operation is more complex. It takes two parameters, a function **f** and a computation **c**. The intended semantics are then that whenever **ask** is used within **c**, the function **f** is applied to the returned value.

```
1 data Reader a k = Ask (a -> k) | Local (a -> a) k k
2
3 ask      = Do $ inj $ Get Pure
4 local f c = Do $ inj $ Local f c (Pure ())
```

Haskell

To show why the **local** operation breaks algebraicity, consider the following computation.

```
1 local (* 2) ask >>= \x -> ask >>= \y -> pure x + y
```

Haskell

Only the first **ask** operation is inside the **local** operation and should therefore be doubled. If the **Reader** effect was algebraic, we should be able to distribute the **>>=** operator again without changing the semantics of the program. However, doing so yields the following computation.

```
1 local (* 2) (ask >>= \x -> ask >>= \y -> pure x + y)
```

Haskell

Now, both **ask** operations are inside the **local** operation, so both values will be doubled. For example, if we had installed a handler that makes **ask** return 1, the first computation would return $2 + 1 = 3$ and the second $2 + 2 = 4$. Therefore, we have shown with a counterexample that the **Reader** effect cannot be algebraic.

A similar argument holds for the **Except** effect, which also has two operations: **catch** and **throw**. In the simplest form, **throw** resembles the **abort** effect, but it takes a parameter that represents an error message. The **catch** operation evaluates its first parameter and jump to the second if it fails, much like the try-catch constructs of languages with effects.

```
1 data Except a k = Throw a | Catch k k
2
3 throw      = Do $ inj $ Throw
4 catch a b = Do $ inj $ Catch a b
```

Haskell

Again, we take a simple example program to show that **Except** violates algebraicity.

```
1 catch (pure False) (pure True) >>= throw -- -> throw False
2 -- then distributing >>= yields
3 catch (throw False) (throw True)          -- -> throw True
```

Haskell

Before distributing the **>>=** operator the computation should throw **False**, but after it should throw **True**. So, again, the semantics have changed by distributing the **>>=** and therefore **Except** is not algebraic.

3.3 The Modularity Problem

Taking a step back from effects, defining a function for exception catching is possible. Recall that the `throw` operation is algebraic, therefore, a handler for it can be defined. If we assume some handler for it called `handleThrow` returns an **Either** where **Left** is the value from `throw` and **Right** is the value from a completed computation, we can define `catch` in terms of that function.

```
1 catch c1 c2 =
2   case handleThrow c1 of
3     Left e -> c2
4     Right a -> pure a
```

Haskell

The distinction between effects which are and which are not algebraic has been described as the difference between *effect constructors* and *effect destructors* (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 using handlers. An imperfect heuristic for whether a function can be an algebraic effect is to check whether the implementation requires a handler. If it uses a handler, it probably cannot be an algebraic effect.

An algebraic effect can have a modular implementation: a computation can be reused in different contexts by using different handlers. For these higher-order effects such as `catch` and `local`, this is not possible. This is known as the *modularity problem* with higher-order effects (Wu, Schrijvers, and Hinze 2014). This is the motivation behind the research on higher-order effects, including this thesis. It is also the problem that the theory of hefty algebras aims to solve.

3.4 Hefty Algebras

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 a solution called hefty algebras. The idea behind hefty algebras is that there is an additional layer of modularity, specifically for higher-order effects.

For a full treatment of hefty algebras, we refer to Bach Poulsen and van der Rest (2023). In addition, the encoding of hefty algebras is explained in more detail by Bach Poulsen (2023).

At the core of hefty algebras are hefty trees. A hefty tree is a generalization of the free monad to higher-order functors, which will write `HOFunctor`. In the listing below, we also repeat the definition of a functor from the previous chapter for comparison.

```
1 -- a regular functor
2 class Functor f where
3   fmap :: (a -> b) -> f a -> f b
4
5 -- a higher-order functor
6 class (forall f. Functor (h f)) => HOFunctor h where
7   hmap :: (f a -> g a) -> (h f a -> h g a)
```

Haskell

The definition of a hefty tree, with the free monad for reference, then becomes:

```

1  -- free monad
2  data Free f a
3    = Pure a
4    | Do (f (Free f a))
5
6  -- hefty tree
7  data Hefty h a
8    = Return a
9    | Do (h (Hefty h) (Hefty h a))

```

Haskell

A hefty tree and a free monad are very similar: we can define the $\gg=$, $<$ and $+$ operators from the previous chapter for hefty trees, so that the hefty tree can be used in the same way.¹ We refer to Bach Poulsen and van der Rest (2023) for the definition of these operators. Furthermore, any functor can be lifted to a higher-order functor with a `Lift` data type.

```

1  data Lift f (m :: * -> *) k = Lift (f k)
2    deriving Functor
3
4  instance Functor f => HOFunctor (Lift f) where
5    hmap _ (Lift x) = Lift x

```

Haskell

In algebraic effects, the evaluation of a computation can be thought of as a transformation of the free monad to the final result:

$$\text{Free } f \ a \xrightarrow{\text{handle}} b.$$

Using hefty algebras, the evaluation instead starts with a *hefty tree*, which is *elaborated* into the free monad. The full evaluation of a computation using hefty algebras then becomes:

$$\text{Hefty } h \ a \xrightarrow{\text{elaborate}} \text{Free } f \ a \xrightarrow{\text{handle}} b.$$

This elaboration is a transformation from a hefty tree into the free monad, defined as an algebra over hefty trees. The algebras are then used in `hfold`; a fold over hefty trees.

```

1  hfold :: HOFunctor h
2    => (forall a. a -> g a)
3    -> (forall a. h g (g a) -> g a)
4    -> Hefty h a
5    -> g a
6  hfold gen _ (Return x) = gen x
7  hfold gen alg (Do x)   =
8    ha alg (fmap (hfold gen alg) (hmap (hfold gen alg) x))
9
10 elab :: HOFunctor h
11    => (forall a. h (Free f) (Free f a) -> Free f a)
12    -> Hefty h a
13    -> Free f a
14 elab elabs = hfold Pure elabs

```

Haskell

For any algebraic – and thus lifted – effect, this elaboration is trivially defined by unwrapping the `Lift` constructor.

¹We are abusing Haskell's syntax here. In the real Haskell encoding, these operators need to have different names from their free monad counterparts, for example :+ and :< .

```

1 | elabLift :: g < f => Lift g (Free f) (Free f a) -> Free f a
2 | elabLift (Lift x) = Op (inj x)

```

Haskell

543

544 Applying `elabLift` to `elab` then gives a function which elaborates `Hefty (Lift f) a` to
 545 `Free f a` for any functor `f`. The more interesting case is that of higher-order effects. For
 546 example, the `local` operation of the `Reader` effect can be mapped to a computation using
 547 the free monad as well, resembling the definition of `local` as a function.

```

1 | data Reader r k = Local r k k
2 |
3 | elabReader :: Ask r < f
4 |             => Reader r (Free f) (Free f a)
5 |             -> Free f a
6 | elabReader (Local f m k) = ask >>= \r -> handle (hAsk (f r)) m >>= k

```

Haskell

548

549 This definition of `elabReader` is modular, because it is a transformation of the computation.
 550 Even if the computation is fixed, the elaboration gives the `local` operation its meaning.
 551 Hence, hefty algebras solve the modularity problem.

FEEDBACK:
Casper: Explain why.

552 These elaborations can be composed to construct elaborations for multiple effects as well.
 553 Bach Poulsen and van der Rest (2023) do this by introducing an operator which composes
 554 elaborations. The composed elaborations are then applied all at once.

555 **TODO:** Mention that the order of elaborations does not matter

FEEDBACK:
Casper: clarify implications.

556 Elaine’s model for higher-order effects, which is explained in Section 4.8, ignores this
 557 constraint. Instead, Elaine allows for elaboration of single higher-order effects at a time. This
 558 means that elaboration cannot return `Free` directly, because `Free` cannot contain higher-order
 559 effects. Instead, we need to work with some other type that encodes both higher-order effects
 560 and the algebraic effects they elaborated into. We will call this type `Comp` for “computation”.
 561 A `Comp h f a` is then equivalent to `Hefty (h + Lift f) a`. The signature of an elaboration
 562 for a higher-order effect `h` would then become the following.

```

1 | handleA :: Free (f + f') a
2 |         -> Free f' a
3 |
4 | elabH    :: Comp (h + h') f a
5 |         -> Comp h' f a

```

Haskell

563

564 What this type `Comp` would be exactly and how this change impacts the rest of the encoding,
 565 is an open question. It is a question we do not need to be concerned with in the design of
 566 Elaine, because we can define semantics that work this way. Nevertheless, it is important to
 567 note, because defining this type correctly, would cement the theory that Elaine is based on.
 568 It is also necessary for writing a formal encoding of its semantics in a language like Agda.
 569 Therefore, it is a gap that future research can fill.

570 We believe this variation to be equivalent to hefty algebras, because higher-order effects
 571 cannot be elaborated into other higher-order effects. Therefore, the elaborations do not
 572 meaningfully interact with each other. In the prototype, this variation has also not presented
 573 any problems.

574 **TODO:** Explain why order of elaborations does not matter

575 Otherwise, Elaine follows the framework of hefty algebras closely. This concludes the
 576 theory that Elaine relies on. Without further ado then, we will dive into the design of Elaine
 577 in the next chapter.

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 implicit elaboration resolution, which is detailed in Chapter 5.

4.1 Overview

At its core, Elaine is based on the lambda calculus, extended with algebraic and higher-order effects. The feature set has been chosen to be comprehensive enough for fairly extensive programs, which are given in Appendix A.

Elaine’s syntax is mostly inspired by Koka (Leijen 2014, 2017) and Rust (Matsakis and Klock 2014). The keywords of the language will be particularly familiar to Rust programmers. It is designed to be relatively simple to parse, which is most clearly reflected in the fact that whitespace is ignored and that there are no infix operators. As a result, Elaine requires semicolons at the end of each statement and requires computations consisting of multiple statements to be wrapped in braces.

All expressions in Elaine are statically and strongly typed with a type system based on Hindley-Milner style type inference (Hindley 1969; Milner 1978). The type system has a special treatment for effect rows similar to Koka’s approach. In most cases, types can be completely inferred and do not need to be specified. Additionally, algebraic data types and tuples are supported for modelling complex data.

Like Koka, Elaine has strict semantics. This means that effects can only occur during function application (Leijen 2014). Additionally, the order in which effects are performed is very clear in this model. We believe this makes effects easier to reason about than in a language with lazy evaluation. Naturally, lazy evaluation can still be encoded into a strict language (Wadler 1996). It also matches the more imperative style Elaine programs are written in. There is currently no way for Elaine programs to interact with the operating system; there is no equivalent to the **IO** monad from Haskell or the `console` and `fsys` effects from Koka.

The source code for the Elaine prototype and additional examples are included in the artefact accompanying this thesis. The full specification for Elaine, including typing judgements and reduction semantics are given in Appendix B.

4.2 Basics

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


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

Elaine

612

613 This example highlights several aspects of Elaine. Comments start with `#` and continue
614 until the end of the line. We bind variables with the `let` keyword. The `main` variable is
615 required, and the value assigned to it is printed at the end of execution. In contrast with
616 other languages, `main` is not a function in Elaine. Note that statements are required to end
617 with a semicolon.

618 In addition to strings, Elaine features integers and booleans as built-in types. To operate
619 on these types, we need functions to perform the operations. By default, there are no functions
620 in scope, however, we can bring them in scope by importing the functions from the `std` module
621 with the `use` keyword. For example, we can write a program that computes $5 \cdot 2 + 3$:

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

Elaine

622

623 The `std` module contains functions for boolean and integer arithmetic, comparison of values,
624 and more. The full list of functions is given in Appendix B.6. To show off some more
625 functions, below is a program that compares the results of two calculations. Note that `-` is
626 allowed as part of an integer literal, but not as an operator. The functions used here are
627 “greater than” (`gt`), exponentiation (`pow`), negation (`neg`), and multiplication (`mul`).

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

Elaine

628

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

```
1 | use std;
2 | let a = pow(2, 5);
3 | let main = {
4 |     let b = mul(25, -30);
5 |     gt(a, neg(b))
6 | };
```

Elaine

634

635 Functions are defined with `fn`, followed by a list of arguments and a function body. Unlike
636 Haskell, functions are called with parentheses. Note that Elaine does not support function
637 currying.

```

1 use std;
2 let double = fn(x) {
3     mul(2, x)
4 };
5 let square = fn(x) {
6     mul(x, x) # or pow(x, 2)
7 };
8 let main = double(square(8));

```

Elaine

638

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

```

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

```

Elaine

641

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

```

1 # The standard library contains basic functions for manipulation
2 # of integers, booleans and strings.
3 use std;
4
5 # Functions are created with `fn` and bound with `let`, just like
6 # other values. The last expression in a function is returned.
7 let square = fn(x: Int) Int {
8     mul(x, x)
9 };
10
11 let is_even = fn(x: Int) Bool {
12     eq(0, modulo(x, 2))
13 };
14
15 # Type annotations can be inferred:
16 let square_is_even = fn(x) {
17     let result = is_even(square(x));
18     if result { "even" } else { "odd" }
19 };
20
21 let give_answer = fn(f, s, x) {
22     let prefix = concat(concat(s, " "), show_int(x));
23     let text = concat(prefix, " is ");
24     let answer = f(x);
25     concat(text, answer)
26 };
27
28 let main = give_answer(square_is_even, "The square of", 4);

```

Elaine

Figure 4.1: A simple Elaine program. The result of this program is the string "The square of 4 is even".

4.3 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. Types and type constructors are capitalized.

The primitive types are `String`, `Bool`, and `Int` for strings, booleans, and integers respectively.

```
1 | let x: Int = 5;           # ok!
2 | let x: String = 5;       # type error!
3 |
4 | let triple = fn(x: Int) Int { mul(3, x) };
5 | let y = triple("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.

```
1 | let triple: fn(Int) Int = fn(x) { mul(3, x) };
```

Type parameters start with a lowercase letter. Like in Haskell, they do not need to be declared explicitly.

```
1 | let f = fn(x: a) (a, Int) {
2 |     (x, 5)
3 | };
4 | let y = f("hello");
5 | let z = f(5);
```

4.4 Algebraic Data Types

Complex programs often require custom 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 declaration 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)`. Since the `Maybe` type is very common, it is provided in the standard library is the `maybe` module.

```

1  use std;
2
3  type Maybe[a] {
4      Just(a),
5      Nothing(),
6  }
7
8  let safe_div = fn(x, y) Maybe[Int] {
9      if eq(y, 0) {
10         Nothing
11     } else {
12         Just(div(x, y))
13     }
14 };
15
16 let main = match safe_div(5, 0) {
17     Just(x) => show_int(x),
18     Nothing => "Division by zero!",
19 };

```

Elaine

669

670 Data types can be recursive. For example, we can define a `List` with a `Cons` and a `Nil`
 671 constructor.

```

1  type List[a] {
2      Cons(a, List[a]),
3      Nil(),
4  }
5
6  let list: List[Int] = Cons(1, Cons(2, Nil()));

```

Elaine

672

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

676 4.5 Recursion & Loops

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

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

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

```

1  use std;
2
3  let rec factorial = fn(x: Int) {
4      if eq(x, 0) {
5          1
6      } else {
7          mul(x, factorial(sub(x,1)))
8      }
9  };

```

Elaine

685

686 A word of caution: Elaine has no guards against unbounded recursion of functions or even
 687 recursive expressions. For example, the statements below are valid according to the Elaine
 688 type checker, but will cause infinite recursion when evaluated, which in practice means that
 689 it will run until the interpreter runs out of memory and crashes.

```

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

```

Elaine

690

691 Using recursive definitions, we can build functions like `map`, `foldl`, and `foldr` to operate on
 692 our previously defined `List` type. The implementation for `map` might look like the listing
 693 below. Note that, in contrast with Haskell, Elaine evaluates these functions eagerly; there is
 694 no lazy evaluation.

```

1  let rec map = fn(f: fn(a) b, list: List[a]) List[b] {
2      match list {
3          Cons(a, as) => Cons(f(a), map(f, list)),
4          Nil() => Nil(),
5      }
6  };
7
8  let doubled = map(fn(x) { mul(2, x) }, [1, 2, 3]); # -> [2, 4, 6]

```

Elaine

695

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

699 4.6 Algebraic Effects

700 The programs in the previous sections are all pure and contain no effects. While a monadic
 701 approach is possible, Elaine provides first class support for algebraic effects and effect handlers
 702 to make working with effects more ergonomic. The design of effects in Elaine is heavily
 703 inspired by Koka (Leijen 2014).

704 An effect is declared with the **effect** keyword. An effect needs a name and a set of
 705 operations. Operations are the functions that are associated with the effect. They can have
 706 an arbitrary number of arguments and a return type. Only the signature of operations can
 707 be given in an effect declaration, the implementation must be provided via handlers (see
 708 Section 4.6.1).

709 Figure 4.2 lists examples of effect declarations for the `Abort`, `Ask`, `State`, and `Write`
 710 effects. We will refer to those declarations throughout this section. For the listings in this
 711 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.

```

1 | effect Abort {
2 |     abort() (),
3 | }

1 | effect Ask {
2 |     ask() Int,
3 | }

1 | effect State {
2 |     get() Int,
3 |     put(Int) (),
4 | }

1 | effect Write {
2 |     write(String) (),
3 | }
```

Figure 4.2: Examples of algebraic effect declarations for some simple effects.

4.6.1 Effect Handlers

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

For example, if we want to use an effect to provide an implicit value, we can make an effect `Ask` and a corresponding handler, which `resumes` 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.

```

1 | let hAsk = handler { ask() { resume(10) } };
2 |
3 | let main = handle[hAsk] add(ask(), ask()); # evaluates to 20
```

Elaine

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.

739 **handle** expression, hence, if we return from the operation, we return from the **handle** ex-
740 pression.

4. A TOUR OF ELAINE

741 The **Abort** effect is an example which does not call the continuation. It defines a single
742 operation **abort**, which stops the evaluation of the computation. The canonical handler for
743 **Abort**, which returns the **Maybe** monad. If the computation returns, it should wrap the
744 returned value in **Just**. Otherwise, if the computation aborts, it should return **Nothing()**.
745 In Elaine, if a sub-computation of a handler returns, the optional **return** arm of the handler
746 will be applied. In the code below, this wraps the returned value in a **Just**. All arms of a
747 handler must have the same return type.

```
1  effect Abort {  
2    abort() a  
3  }  
4  
5  let hAbort = handler {  
6    return(x) { Just(x) }  
7    abort() { Nothing() }  
8  };  
9  
10 let main = handle[hAbort] {  
11   abort();  
12   5  
13 };
```

Elaine

748

749 Alternatively, we can define a handler that defines a default value for the computation in
750 case it aborts. This is more convenient than the first handler if the **abort** case should always
751 become

```

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

```

Elaine

767

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. 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*.

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

$$\langle A_1, \dots, A_N | e \rangle = \langle B_1, \dots, B_M | f \rangle,$$

for `e` and `f`. To do so, a fresh variable `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.

4.7 Functions Generic over Effects

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.

```

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

```

Elaine

788

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.

```

1 | let main: <> Maybe[()] = handle[hAbort] {
2 |     abort_on_false(fn() { false })
3 | };

```

Elaine

793

Recall the definition of `map` in Section 4.5, which was written without any effects in its signature. Adding the effects yields the following definition.

```

1 | let rec map = fn(f: fn(a) <|e> b, l: List[a]) <|e> List[b] {
2 |     match l {
3 |         Nil() => Nil(),
4 |         Cons(x, xs) => Cons(f(x), map(f, xs)),
5 |     }
6 | };

```

Elaine

796

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.


```

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

```

Elaine

If we were to 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.

Functional languages like Haskell usually do not feature a construct for looping. This is partly because folds, maps, and recursion are preferred to loops, but also because a looping construct relies on effects. In Koka and Elaine, we can define a `while` function which is generic over effects. This enables both functional and imperative styles of programming.

TODO: Check

```

1 | let rec while = fn(
2 |   predicate: fn() <|e> Bool,
3 |   body: fn() <|e> ()
4 | ) <|e> () {
5 |   if predicate() {
6 |     ()
7 |   } else {
8 |     body()
9 |     while(predicate, body)
10 |   }
11 | };

```

Elaine

4.8 Higher-Order Effects

Higher-order effects in Elaine are supported via elaborations, as proposed by Bach Poulsen and van der Rest (2023) and explained in Section 3.4. 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` operation 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. This effect enhances the previously introduced `Ask` effect with a `local` operation that modifies the value returned by `ask`. To motivate the implementation, let us first imagine how to emulate the behaviour of `local`. Our goal is to make the following snippet return the value 15.

```

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

```

Elaine

835

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

```

1 let local = fn(f: fn(Int) Int, g: fn() <Ask|e> a) <|e> a {
2   handle[hAsk(f(ask()))] { g() }
3 }

```

Elaine

839

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

```

1 effect Reader! {
2   local!(fn(Int) Int, a) a
3 }

```

Elaine

843

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

```

1 let eLocal = elaboration Reader! -> <Ask> {
2   local!(f, c) {
3     handle[hAsk(f(ask()))] c
4   }
5 }

```

Elaine

849

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

```

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

```

Elaine

856

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

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

```

1  type Result[a, b] {
2      Ok(a),
3      Err(b),
4  }
5
6  effect Throw {
7      throw(String) a
8  }
9
10 let hThrow = handler {
11     return(x) { Ok(x) }
12     throw(s) { Err(s) }
13 };

```

861

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

```

1  effect Exception! {
2      throw!(String) a
3      catch!(a, a) a
4  }

```

866

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

```

1  let eExcept = elaboration Exception! -> <Throw> {
2      throw!(s) { throw(s) }
3      catch!(a, b) {
4          match handle[hThrow] a {
5              Ok(x) => x,
6              Err(s) => b,
7          }
8      }
9  };

```

872

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

```

1 let main = handle[hThrow] elab[eExcept] {
2   let x = 0;
3   catch!{
4     if eq(x, 0) {
5       throw!("Whoa, x can't be zero!")
6     } else {
7       sub(x, 1)
8     }
9   }, 0)
10 };

```

Elaine

876

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

```

1 let eExceptLog = elaboration Exception! -> <Throw,Log> {
2   throw!(s) { throw(s) }
3   catch!(a, b) {
4     match handle[hThrow] a {
5       Ok(x) => x,
6       Err(s) => {
7         log(s);
8         b
9       }
10    }
11  }
12 };

```

Elaine

880

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

```

1 let eExceptIgnoreCatch = elaboration Exception! -> <Throw> {
2   throw!(s) { throw(s) }
3   catch!(a, b) { a }
4 }

```

Elaine

886

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

Chapter 5

Implicit Elaboration Resolution

With Elaine, we aim to explore further ergonomic improvements 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 the language should be able to infer the elaborations. Take the example in the listing below, where we let Elaine infer the elaboration for us.

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 apply them in arbitrary order.

IMPLICIT ELABORATION RESOLUTION

A nice property of this transformation is that it 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**]. A code editor could then display this transformed **elab** as an inlay hint.

Chapter 6

Related Work

This chapter discusses extensions to algebraic effects and alternatives to algebraic effects and hefty algebras. Additionally, we discuss some other languages with effects and the various alternative syntax and semantics.

6.1 Monad Transformers

Monad transformers provide a way to compose monads (Moggi 1989). This makes them an alternative to the free monad. While monad transformers predate algebraic effects, they do support higher-order effects. A popular implementation of monad transformers is Haskell’s `mtl`¹ library. In the rest of this section, we adopt the terminology from that library.

The goal of monad composition is to make the operations of all composed monads available to the computation. Given two monads `A` and `B`, a naive composition would result in the type `A (B a)`. However, this type represents a computation using `A` that returns a computation `B a`, meaning that it is not possible to use operations of both monads.

A monad transformer is a type constructor that takes some monad and returns a new monad. Usually, the transformation it performs is to add operations to the input monad. Composing `A` and `B` then requires some transformer `AT` to be defined, such that `AT B` is a monad that provides the operations of both `A` and `B`. An arbitrary number of monad transformers can be composed this way. The representation of a monad then becomes much like that of a list of monad transformers. The `Identity` monad marks the end of the list, and is defined as below.

```
1 | newtype Identity a = Identity a
```

Haskell

A neat property of monad transformers is that a monad can be easily obtained by applying the transformer to the identity monad. Haskell’s `mtl` library, for instance, defines a monad transformer `StateT` and then defines `State` as `StateT Identity`. The operations of the state effect are then not implemented on `StateT` directly, but on are part of a type class `MonadState`. The `StateT` is then an instance of `MonadState` class. Every other transformer is an instance of `MonadState` if its input monad is an instance of `MonadState`. For example, for the `WriterT` instance, there is the following instance declaration.

¹<https://github.com/haskell/mtl>


```

1 instance MonadState s (StateT s m) where
2   -- definitions omitted
3
4 instance MonadState s m => MonadState s (WriteT m) where
5   -- definitions omitted

```

Haskell

A computation can then be generic over the monad transformers, requiring only that `StateT` is present somewhere in the stack of monad transformers.

```

1 usesState :: MonadState Int m => Int -> m Int
2 usesState a = get >>= \x -> put (x + a)

```

Haskell

This is analogous to the `State s < f` constraint from the free monad encoding. However, there is a cost to this approach. For every effect, a new type class needs to be introduced and there need to be instance definitions on all existing monad transformers. The number of instance declarations therefore scales quadratically with the number of effects.

Another downside to monad transformers is that the order in which the monads need to be evaluated is entirely fixed. In the free monad encoding and languages with algebraic effects, the effects in the effect row can be reordered. **The order of the monad transformers also determines the order in which they must be handled: the outermost monad transformer must be handled first.**

FEEDBACK
translate handling to the context of monad transformers

6.2 Other Solutions to the Modularity Problem

An alternative to hefty algebras for solving the modularity problem is the theory of *scoped effects* (Piróg et al. 2018; Wu, Schrijvers, and Hinze 2014; Yang et al. 2022). This theory replaces the free monad by a `Prog` monad, which features one additional constructor called `Enter`. Along with the continuation, this constructor takes a sub-computation. The return value of this sub-computation is passed to the continuation. In that sense, the `Enter` constructor matches the `>>=`, but without distributing the continuation over its sub-computation.

Instead of defining evaluation as a single algebra, scoped effects requires two algebras: an endo-algebra for scoped operations and a base-algebra for the other operations. This is somewhat similar to the distinction between elaboration and handling for hefty algebras, however, in hefty algebras, the algebras are not applied at the same time.

Many higher-order effects, such as the exception and reader effects, can be expressed in this framework. However, it is less general than hefty algebras, because there are some higher-order effects that cannot be expressed as scoped effects. This concerns effects that defer some computation, such as the lambda abstraction (van den Berg et al. 2021). Hefty algebras are therefore more general than scoped effects (Bach Poulsen and van der Rest 2023).

The limitations of scoped effects can be understood intuitively by emulating them in Elaine. The endo-algebra of scoped effects corresponds roughly with a **handle** operation in an elaboration. Since the result of the sub-computation in scoped effect must directly be passed to the continuation, the elaboration contains only a **handle** and nothing else. Therefore, any higher-order effect that can be expressed as the elaboration below (up to renaming) can be defined in the theory of scoped effects. However, this is an informal and imperfect comparison, since scoped effects and hefty algebras have very different semantics.

```

1 | effect ScopedEffect! {
2 |     scoped_operation!(a) a
3 | }
4 |
5 | let eScoped = elaboration ScopedEffect! -> AlgebraicEffect {
6 |     scoped_operation!(a) {
7 |         handle[endoAlg] a
8 |     }
9 | };

```

Elaine

Scoped effects have been generalized by van den Berg et al. (2021) to *latent effects*, which supports the same set of effects as hefty algebras. Bach Poulsen and van der Rest (2023) note that while latent effects are powerful, they require *weaving glue* to ensure unhandled operations are treated correctly through sub-computations. In contrast, hefty algebras do not require any weaving.

6.3 Languages with First-Class Effects

TODO: Define first-class effects much earlier in the thesis

The motivation of adding support for effects to a programming language is twofold. First, it enables effects to be implemented into languages with type systems in which effects cannot be encoded as a free monad or a similar model. 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.

Notable examples of languages with first-class support for algebraic effects are Koka (Leijen 2014), Frank (Lindley, McBride, and McLaughlin 2017), Effekt (Bach Poulsen and van der Rest 2023), Eff (Bauer and Pretnar 2015), Helium (Biernacki et al. 2019), and OCaml (Sivaramakrishnan et al. 2021). In all of these languages, effect row variables can be used to abstract over effects. For example, the signature of the `map` function in Koka is given below and is similar to the signature of `map` in Elaine.

```

1 | fun map ( xs : list<a>, f : a -> e b ) : e list<b>
2 |     ...

```

Koka

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*. Any effect row then becomes not the exact set of effects that need to be handled, but the smallest set. The equivalent signature of `map` is then written as

```

1 | map : {X -> []Y} -> List X -> []List Y

```

Frank

In contrast with Elaine, languages such as Koka and Frank do not have dedicated types for handlers and **handle** constructs. Instead, they represent handlers as functions that take computations as arguments. In Elaine, there are dedicated types and constructs for effect handlers so that they are symmetric with elaborations. That is, the counterpart of **elab** is **handle** and the counterpart of **elaboration** is **handler**.

Koka implements several extensions to standard algebraic effects. First, it supports named handlers (Xie et al. 2022), which provide a mechanism to distinguish between multiple occurrences of an effect in an effect row. Additionally, Koka features *scoped handlers*, which are

different from the previously mentioned scoped effects. Scoped handlers make it possible to associate types with handler instances (Xie et al. 2022).

6.4 Effects as Free Monads

There are many libraries that implement the free monad in various forms in Haskell, including `fused-effects`², `polysemy`³, `freer-simple`⁴, and `eff`⁵. Each of these libraries give the encoding of effects a slightly different spin in an effort to find the most ergonomic and performant representation. They are all not just based on the free monad, but on freer monads (Kiselyov and Ishii 2016) and fused effects (Wu and Schrijvers 2015) which yield better performance than the free monad. Some of these libraries support scoped effects as well, but apart from the work by Bach Poulsen and van der Rest (2023), no libraries with support for hefty algebras have been published.

Effect rows are often constructed using the *Data Types à la Carte* technique (Swierstra 2008), which requires a fairly robust type 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 in production is strongly discouraged by the author.

The programming language Idris (Brady 2013) also has an implementation of algebraic effects in its standard library. It is an interesting case study since Idris is a dependently typed language. Due to its dependent typing, it can distinguish multiple occurrences of a single effect in the same effect row by assigning them different *labels*. This is similar to what *named handlers* (Xie et al. 2022) aims to accomplish.

²<https://github.com/fused-effects/fused-effects>

³<https://github.com/polysemy-research/polysemy>

⁴<https://github.com/lexi-lambda/freer-simple>

⁵<https://github.com/hasura/eff>

⁶<https://github.com/rosefromthedeath/effing-mad>

Chapter 7

Conclusion

The use of algebraic effects is slowly breaking through from research to mainstream languages. We hope that this thesis contributes to this adoption, by presenting a language that is complete enough to give an impression of what a production-ready language with support for higher-order effects. Elaine is far from production-ready, but it allows for remarkably complex programs to be expressed, making it a good playground to experiment with programming with (higher-order) effects.

We have presented a full language specification and prototype based on hefty algebras. Our focus in this endeavour was to show the viability and explore the ergonomics of such a language. This shows that elaborations are a viable concept for languages with effect systems. The result is, in our opinion, an expressive language in which higher-order effects can be represented with relative ease.

The specification shows how the theory of hefty algebras maps to the syntax and semantics of a programming language. In particular, we have defined typing rules and reduction semantics for elaborations. We also argue that implicit elaboration resolution is a useful feature for a language based on hefty algebras, because it reduces the syntactic overhead of elaborations. Of particular interest is how this feature interacts with the module system for any language, to allow effects to be imported along with their elaborations.

The examples throughout this thesis and in Appendix A also motivate why support for higher-order effects can be a useful, since we can easily define modular operations that languages without higher-order effects can only express as functions.

The semantics of Elaine are slightly different from the theory of hefty algebras, since Elaine does not require all elaborations to be applied at once. We believe this is sound, and it has not presented any problems in the prototype. However, there is no formal argument for this claim. Future work could fill this gap by generalizing hefty algebras such that it allows for multiple separate elaborations.

A missing feature in Elaine is type parameters for effects. In Koka, for example, the state effect `state<s>` is parametrized by a type `s`. We believe Elaine could be extended to support this, however, both the specification and the prototype do not include this feature yet. Another omission are IO operations. An Elaine program cannot write to files, accept input or print text apart from the value it returns. Furthermore, Elaine does not include any extensions of algebraic effects, such as named handlers.

The prototype for Elaine only features an interpreter, not a compiler. So, another direction for future work is towards efficient compilation of elaborations. In other words, transforming a program with elaborations to a program that only uses algebraic effects. Since compilation of algebraic effects is well-established (Leijen 2017), this should enable full compilation of program with higher-order effects.

FEEDBACK:
Casper:
What is
sound?

Bibliography

- 1098 Bach Poulsen, Casper (2023). *Algebras of Higher-Order Effects in Haskell*. URL: <http://casperbp.net/posts/2023-08-algebras-of-higher-order-effects/> (visited on
1099 09/09/2023).
- 1101 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).
- 1105 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).
- 1109 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).
- 1115 Biernacki, Dariusz et al. (Jan. 2, 2019). “Abstracting algebraic effects”. In: *Proceedings of the ACM on Programming Languages* 3 (POPL), pp. 1–28. ISSN: 2475-1421. DOI: 10.1145/3290319. URL: <https://dl.acm.org/doi/10.1145/3290319> (visited on 09/16/2023).
- 1118 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).
- 1124 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).
- 1128 Hindley, R. (Dec. 1969). “The Principal Type-Scheme of an Object in Combinatory Logic”. In: *Transactions of the American Mathematical Society* 146, p. 29. ISSN: 00029947. DOI: 10.2307/1995158. URL: <https://www.jstor.org/stable/1995158?origin=crossref> (visited on 09/16/2023).
- 1132 Hutton, Graham and Erik Meijer (1996). *Monadic parser combinators*. URL: <https://nottingham-repository.worktribe.com/output/1024440>.
- 1134 Kiselyov, Oleg and Hiromi Ishii (Jan. 28, 2016). “Freer monads, more extensible effects”. In: *ACM SIGPLAN Notices* 50.12, pp. 94–105. ISSN: 0362-1340, 1558-1160. DOI: 10.1145/

- 1136 2887747.2804319. URL: <https://dl.acm.org/doi/10.1145/2887747.2804319>
1137 (visited on 06/03/2023).
- 1138 Leijen, Daan (July 23, 2005). “Extensible records with scoped labels”. In.
1139 — (June 5, 2014). “Koka: Programming with Row Polymorphic Effect Types”. In: *Electronic Proceedings in Theoretical Computer Science* 153, pp. 100–126. ISSN: 2075-2180.
1140 DOI: 10.4204/EPTCS.153.8. URL: <http://arxiv.org/abs/1406.2061v1> (visited on
1141 06/16/2023).
- 1142 — (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:
1143 The 44th Annual ACM SIGPLAN Symposium on Principles of Programming Languages.
1144 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
1145 04/08/2023).
- 1146 — (Mar. 15, 2023). *The Koka Programming Language*. URL: <https://koka-lang.github.io/koka/doc/book.html> (visited on 06/13/2023).
- 1147 Lindley, Sam, Conor McBride, and Craig McLaughlin (Oct. 3, 2017). *Do be do be do*. arXiv:
1148 1611.09259[cs]. URL: <http://arxiv.org/abs/1611.09259> (visited on 04/08/2023).
- 1149 Matsakis, Nicholas D. and Felix S. Klock (Oct. 18, 2014). “The rust language”. In: *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology*. HILT
1150 ’14: High Integrity Language Technology ACM SIGAda Annual Conference. Portland Ore-
1151 gon USA: ACM, pp. 103–104. ISBN: 978-1-4503-3217-0. DOI: 10.1145/2663171.2663188.
1152 URL: <https://dl.acm.org/doi/10.1145/2663171.2663188> (visited on 09/16/2023).
- 1153 Milner, Robin (Dec. 1978). “A theory of type polymorphism in programming”. In: *Journal of Computer and System Sciences* 17.3, pp. 348–375. ISSN: 00220000. DOI: 10.1016/
1154 0022-0000(78)90014-4. URL: [https://linkinghub.elsevier.com/retrieve/pii/
1155 0022000078900144](https://linkinghub.elsevier.com/retrieve/pii/0022000078900144) (visited on 09/16/2023).
- 1156 Moggi, Eugenio (1989). *An Abstract View of Programming Languages*.
- 1157 Peyton Jones, Simon L. and Philip Wadler (1993). “Imperative functional programming”. In:
1158 *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of program-
1159 ming languages - POPL ’93*. the 20th ACM SIGPLAN-SIGACT symposium. Charleston,
1160 South Carolina, United States: ACM Press, pp. 71–84. ISBN: 978-0-89791-560-1. DOI:
1161 10.1145/158511.158524. URL: [http://portal.acm.org/citation.cfm?doid=
1162 158511.158524](http://portal.acm.org/citation.cfm?doid=158511.158524) (visited on 06/03/2023).
- 1163 Piróg, Maciej et al. (July 9, 2018). “Syntax and Semantics for Operations with Scopes”. In:
1164 *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*.
1165 LICS ’18: 33rd Annual ACM/IEEE Symposium on Logic in Computer Science. Oxford
1166 United Kingdom: ACM, pp. 809–818. ISBN: 978-1-4503-5583-4. DOI: 10.1145/3209108.3209166. URL: <https://dl.acm.org/doi/10.1145/3209108.3209166> (visited on
1167 09/11/2023).
- 1168 Plotkin, Gordon and John Power (2001). “Adequacy for Algebraic Effects”. In: *Foundations of
1169 Software Science and Computation Structures*. Ed. by Furio Honsell and Marino Miculan.
1170 Red. by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen. Vol. 2030. Series Title:
1171 Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 1–
1172 24. ISBN: 978-3-540-41864-1 978-3-540-45315-4. DOI: 10.1007/3-540-45315-6_1. URL:
1173 http://link.springer.com/10.1007/3-540-45315-6_1 (visited on 04/08/2023).
- 1174 — (2003). “Algebraic Operations and Generic Effects”. In: *Applied Categorical Structures*
1175 11.1, pp. 69–94. ISSN: 09272852. DOI: 10.1023/A:1023064908962. URL: [http://link.
1176 springer.com/10.1023/A:1023064908962](http://link.springer.com/10.1023/A:1023064908962) (visited on 06/03/2023).
- 1177 Plotkin, Gordon and Matija Pretnar (2009). “Handlers of Algebraic Effects”. In: *Programming
1178 Languages and Systems*. Ed. by Giuseppe Castagna. Vol. 5502. Series Title: Lecture Notes
1179 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).
- Sivaramakrishnan, K. C. et al. (June 19, 2021). “Retrofitting Effect Handlers onto OCaml”. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pp. 206–221. DOI: 10.1145/3453483.3454039. arXiv: 2104.00250[cs]. URL: <http://arxiv.org/abs/2104.00250> (visited on 04/08/2023).
- Swierstra, Wouter (July 2008). “Data types à la carte”. In: *Journal of Functional Programming* 18.4. ISSN: 0956-7968, 1469-7653. DOI: 10.1017/S0956796808006758. URL: http://www.journals.cambridge.org/abstract_S0956796808006758 (visited on 06/11/2023).
- 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).
- (June 1996). “Lazy versus strict”. In: *ACM Computing Surveys* 28.2, pp. 318–320. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/234528.234738. URL: <https://dl.acm.org/doi/10.1145/234528.234738> (visited on 09/16/2023).
- Wu, Nicolas and Tom Schrijvers (2015). “Fusion for Free: Efficient Algebraic Effect Handlers”. In: *Mathematics of Program Construction*. Ed. by Ralf Hinze and Janis Voigtländer. Vol. 9129. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, pp. 302–322. ISBN: 978-3-319-19796-8 978-3-319-19797-5. DOI: 10.1007/978-3-319-19797-5_15. URL: https://link.springer.com/10.1007/978-3-319-19797-5_15 (visited on 09/11/2023).
- Wu, Nicolas, Tom Schrijvers, and Ralf Hinze (June 10, 2014). *Effect Handlers in Scope*.
- Xie, Ningning et al. (Oct. 31, 2022). “First-class names for effect handlers”. In: *Proceedings of the ACM on Programming Languages* 6 (OOPSLA2), pp. 30–59. ISSN: 2475-1421. DOI: 10.1145/3563289. URL: <https://dl.acm.org/doi/10.1145/3563289> (visited on 06/13/2023).
- Yang, Zhixuan et al. (2022). “Structured Handling of Scoped Effects: Extended Version”. In: Publisher: arXiv Version Number: 1. DOI: 10.48550/ARXIV.2201.10287. URL: <https://arxiv.org/abs/2201.10287> (visited on 06/13/2023).

1219 Appendix A

1220 Elaine Example Programs

1221 This chapter contains longer Elaine samples with some additional explanation.

1222 A.1 A naive SAT solver

1223 This program is a naive brute-forcing SAT solver. We first define a `Yield` effect, so we can
1224 yield multiple values from the computation. We will use this to find all possible combinations
1225 of boolean inputs that satisfy the formula. The `Logic` effect has two operations. The `branch`
1226 operation will call the continuation twice; once with **false** and once **true**. With `fail`, we
1227 can indicate that a branch has failed. To find all solutions, we just `branch` on all inputs and
1228 `yield` when a correct solution has been found and `fail` when the formula is not satisfied.
1229 In the listing below, we check for solutions of the equation $\neg a \wedge b$.

1231 **A.2 The Reader Effect**

1232 **TODO:** explain

```

14 }
15
16 let hOut = handler {
17   return(x) { Output("", x) }
18   tell(s) {
19     match resume(()) {
20       Output(s', x) => Output(concat(s, s'), x)
21     }
22   }
23 };
24
25 let eWriter = elaboration Writer! -> <Out> {
26   tell!(s) { tell(s) }
27   censor!(f, c) {
28     match handle[hOut] c {
29       Output(s, x) => {
30         tell(f(s));
31         x
32       }
33     }
34   }
35 };
36
37 let main = handle[hOut] elab {
38   tell("foo");
39   censor!(fn(s) { "bar" }, {
40     tell("baz");
41     5
42   });
43 };

```

1235

1236 A.4 Structured Logging

1237

TODO: explain

1239 A.5 Parser Combinators

1240 Monadic parser combinators (Hutton and Meijer 1996) are a popular technique for construct-
1241 ing parsers. The parser for Elaine is also written using `megaparsec`¹, which is a monadic
1242 parser combinator library for Haskell. Attempts have been made to implement parser combi-
1243 nators using algebraic effects. However, it requires higher-order combinators for a full feature
1244 set matching that of monadic parser combinators. For example, the `alt` combinator takes
1245 two branches and attempts to parse the first branch and tries the second branch if the first
1246 one fails. This is remarkably similar to the catch operation of the exception effect and is
1247 indeed higher-order.

1248 Below is a full listing of a JSON parser written in Elaine using a variation on parser
1249 combinators using effects. It is implemented using a higher-order `Parse!` effect, which is
1250 elaborated into a state and an abort effect, which are imported from the standard library.
1251 The `try!` effect is a higher-order effect which takes an effectful computation as an argument.

1252 Higher-order effects are convenient for parser combinators, but not necessary. There is a
1253 parsing effect in a more algebraic style written in Effekt available at [https://effekt-lang.](https://effekt-lang.org/docs/casestudies/parser)
1254 [org/docs/casestudies/parser](https://effekt-lang.org/docs/casestudies/parser)

¹<https://github.com/mrkkrrp/megaparsec>

```

69         c
70     } else {
71         fail!()
72     }
73 };
74
75 let rec contains_str = fn(s, l) {
76     match l {
77         Cons(x, xs) => {
78             if str_eq(x, s) {
79                 true
80             } else {
81                 contains_str(s, xs)
82             }
83         },
84         Nil() => false,
85     }
86 };
87
88 let one_of = fn(s) {
89     let list_of_chars = explode(s);
90     fn() {
91         let c = eat!();
92         if contains_str(c, list_of_chars) {
93             c
94         } else {
95             fail!()
96         }
97     }
98 };
99
100 # Parse a single digit
101 let digit = one_of("0123456789");
102 let str_char = one_of(join([
103     "0123456789",
104     "ABCDEFGHIJKLMNOPQRSTUVWXYZ",
105     "abcdefghijklmnopqrstuvwxyz",
106     "_- ?!",
107 ]));
108 let white_one = one_of(" \n\t");
109 let white = fn() { many(white_one); () };
110
111 let tokenws = fn(s) {
112     let t = token(s);
113     white();
114     t
115 };
116
117 let comma_separated = fn(p: fn() <Parse!> a) <Parse!> List[a] {
118     separated(p, fn() { tokenws(",") })
119 };
120
121 # Parse as many digits as possible
122 let number = fn() { join(many(digit)) };
123
124 type Json {

```

1255

```

125     jsonString(String),
126     jsonInt(String),
127     jsonArray(List[Json]),
128     jsonObject(List[(String, Json)]),
129 }
130
131 let string = fn() <Parse!> String {
132     token("\"");
133     let s = join(many(str_char));
134     tokenws("\"");
135     s
136 };
137
138 let key_value = fn(value: fn() <Parse!> Json) <Parse!> (String, Json) {
139     let k = string();
140     tokenws(":");
141     (k, value())
142 };
143
144 let object = fn(value: fn() <Parse!> Json) <Parse!> Json {
145     tokenws("{");
146     let kvs = comma_separated(fn() { key_value(value) });
147     tokenws("}");
148     jsonObject(kvs)
149 };
150
151 let array = fn(value) {
152     tokenws("[");
153     let values = comma_separated(value);
154     tokenws("]");
155     jsonArray(values)
156 };
157
158 let rec value = fn() {
159     alt([
160         fn() { array(value) },
161         fn() { object(value) },
162         fn() { jsonString(string()) },
163         fn() { jsonInt(number()) },
164     ])
165 };
166
167 let parse = fn(parser, input) {
168     let f = handle[hState] handle[hAbort] elab[eParse] parser();
169     f(input)
170 };
171
172 let main = parse(
173     value,
174     "{\"key1\": 123, \"key2\": [1,2,3], \"key3\": \"some string\"}"
175 );

```

1256

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

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.

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

Elaine

program $p ::= d \dots d$

declaration $d ::= [\text{pub}] \text{ mod } x \{d \dots d\}$
| $[\text{pub}] \text{ use } x;$
| $[\text{pub}] \text{ let } [\text{rec}] p = e;$
| $[\text{pub}] \text{ effect } \phi \{s, \dots, s\}$
| $[\text{pub}] \text{ type } x \{s, \dots, s\}$

block $b ::= \{ es \}$

expression list $es ::= e; es$
| $\text{let } [\text{rec}] p = e; es$
| e

expression $e ::= x$
| $()$ | true | false | number | string
| $\text{fn}(p, \dots, p) [T] b$
| $\text{if } e b \text{ else } b$
| $e(e, \dots, e)$ | $\phi(e, \dots, e)$
| $\text{handler } \{\text{return}(x) b, o, \dots, o\}$
| $\text{handle}[e] e$
| $\text{elaboration } x! \rightarrow \Delta \{o, \dots, o\}$
| $\text{elab}[e] e$ | $\text{elab } e$
| es

annotatable variable $p ::= x : T$ | x

signature $s ::= x(T, \dots, T) T$

effect clause $o ::= x(x, \dots, x) b$

type $T ::= \Delta \tau$ | τ

value type $\tau ::= x$
| $()$ | Bool | Int | String
| $\text{fn}(T, \dots, T) T$
| $\text{handler } x \tau \tau$
| $\text{elaboration } x! \Delta$

effect row $\Delta ::= \langle \phi, \dots, \phi[|x] \rangle$

effect $\phi ::= x$ | $x!$

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:

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

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

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

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

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

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

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

TODO: Talk about (Leijen 2005, 2014).

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

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

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

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

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

1325 we assert that $A \subseteq B$. If that is true, we can substitute e_n for the effects in $B - A$.

1326 Finally, there is the case where both rows are open:

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

1327 In this case, unification is always possible, because both rows can be extended with the effects
1328 of the other. We create a fresh effect row variable e_C with the following substitutions:

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

1329 In other words, A is extended with the effects that are in B but not in A and, similarly, B is
1330 extended with the effects in A but not in A .

1331 B.3 Typing judgments

1332 The context $\Gamma = (\Gamma_M, \Gamma_V, \Gamma_E, \Gamma_\Phi)$ consists of the following parts:

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

INFO: A 1333
 Γ_T for data
types might
be added.

1334 Whenever one of these is extended, the others are implicitly passed on too, but when
1335 declared separately, they not implicitly passed. For example, Γ'' is empty except for the
1336 single $x : T$, whereas Γ' implicitly contains Γ_M, Γ_E & Γ_Φ .

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

1337 If the following invariants are violated there should be a type error:

- 1338 • The operations of all effects in scope must be disjoint.
- 1339 • Module names are unique in every scope.
- 1340 • Effect names are unique in every scope.

1341 B.3.1 Type inference

1342 We have the usual generalize and instantiate rules. But, the “generalize” rule requires an
empty effect row.

QUESTION
Koka requires
an empty
effect row.
Why?

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

1345 Where ftv refers to the free type variables in the context.

B.3.2 Expressions

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

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

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

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

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

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

B.3.3 Declarations and Modules

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

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

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

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

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

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

B.3.4 Algebraic Effects and Handlers

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

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

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

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

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

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

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

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

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

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

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

1380 B.4 Desugaring

1381 To simplify the reduction rules, we simplify the AST by desugaring some constructs. This
1382 transform is given by a fold over the syntax tree with the following operation:

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

1383 B.5 Semantics

1384 The semantics of Elaine are defined as reduction semantics.

1385 We use two separate contexts to evaluate expressions. The E context is for all constructs
1386 except effect operations, such as **if**, **let**, and function applications. The X_{op} context is the
1387 context in which a handler can reduce an operation op .

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

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

1388

TODO: Add some explanation

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

1389

B.6 Standard Library

1390 Elaine does not include any operators. This choice was made to simplify parsing of the
 1391 language. For the lack of operators, any manipulation of primitives needs to be done via the
 1392 standard library of built-in functions.

1393 These functions reside in the `std` module, which can be imported like any other module
 1394 with the `use` statement to bring its contents into scope.

1395 The full list of functions available in the `std` module, along with their signatures and
 1396 descriptions, is given in Figure B.2.

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

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