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

Terts Diepraam

September 16, 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`

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

Contents

Chapter 1

Introduction

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

Historically, programming languages have supported effects in different ways. Some programming languages opt to give the programmer virtually unrestricted access to effectful operations. For instance, any part of a C program can interact with memory, the filesystem or the network. The program can even yield control to any location in program with the `goto` keyword, which has famously been criticized by Dijkstra (1968). This core of his argument is that `goto` breaks the structure of the code. The programmer then has to trace the execution of the program in their mind in order to understand it. The same reasoning extends to other effects: the more effects a function is allowed to exhibit, the harder it becomes to reason about.

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

A solution is to treat 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 then also part of the type system, so that it is easier to track which functions are allowed to throw exceptions. However, this means that any effect must be backed by the language. 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 allow.

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 dictating 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

[1]Usually 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.

written in Koka, a functional language. In this program, there is a set of users that are considered administrators and have therefore more privileges. The `all_admins` function checks whether all user ID's in a list are administrators.

```koka
val admins = [0,1,2]

fun is_admin(user_id: int): bool
  admins.any(fn(x) x == user_id)

fun all_admins(l: list<int>): bool
  l.map(is_admin).foldl(True, (&&))

fun main()
  val result = all_admins([0,1,2,3])
```

This a fairly standard functional program where the result is a single boolean. However, the program does not tell us which users are not admins. That could be useful information to print. In an imperative language, we could just add a `print` call in `is_admin` and call it a day. But in a functional language, the messages need to be returned from each function that needs to log. These messages then need to be concatenated to build up a string of messages.

```koka
fun is_admin(user_id: int): (bool, string)
  if admins.any(fn(x) x == user_id)
  then (True, "")
  else (False, "Denied " ++ show(user_id) ++ "\n")

fun all_admins(list: list<int>): (bool, string)
  match list
    Nil() -> (True, "")
    Cons(x, xs) ->
      val (y, s) = is_admin(x)
      val (ys, s') = all_admins(xs)
      (y && ys, s ++ s')

fun main()
  val (result, log) = all_admins([1,2,3,4])
```

So, with the logging effect, the program becomes much more involved than the original program. For larger programs, 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 an entirely 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`.

```koka
alias log<a> = (v: a, msg: string)

fun pure(v: a): log<a>
  (v, "")

fun bind(m: log<a>, k: a -> log<b>): log<b>
  val (a, s) = m
  val (b, s') = k(a)
  (b, s ++ s')

fun log(msg: string): log<()>
  ((), msg)
```

In a language like Haskell, this definition can be encoded in a type class, so that a function can be generic over all monads. The `is_admin` and `all_admins` can then be written using these functions instead of dealing with the strings in the tuples directly. Hence, we have abstracted over the effect and could replace it with another.

```koka
fun all_admins(list)
  match list
    Nil() -> pure(True)
    Cons(x, xs) -> is_admin(x).bind fn(y)
      all_admins(xs).bind fn(ys)
        pure(y && ys)

fun is_admin(user_id: int): log<bool>
  if admins.any(fn(x) x == user_id)
  then pure(True)
  else
    log("Denied " ++ show(user_id) ++ "\n").bind fn(())
      pure(False)
```

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

Another limitation of the monad approach becomes apparent when we want to use multiple effects. The problem is that the composition of two monads does not yield a monad. This is a limitation that can be worked around with *monad transformers*. A monad transformer takes a monad and returns a monad with additional operations. The operations of every effect then need to be implemented on every transformer. Adding a single effect therefore requires additional implementations of its operations every other monad transformer. The number of implementations therefore grows quadratically with the number of effects.

Instead, we turn to *algebraic effects*. Languages with support for algebraic effects allow effects to be defined modularly. An effect consists of a set of *effect operations*. A computation using an effect then needs to be wrapped in a *handler*, which defines the semantics for the operations. These modular effects and handlers are based on the free monad. Examples of language with algebraic effects include Koka (Leijen 2014), Eff (Bauer and Pretnar 2015), Frank (Lindley, McBride, and McLaughlin 2017), and Effekt (Brachthäuser, Schuster, and Ostermann 2020).

In the listing below, we first declare the effect `log` with a single operation with the same name. This operation takes a message to log. 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.

```koka
effect log
  ctl log(msg: string): ()

val hLog = handler
  return(x) (x, "")
  ctl log(msg)
    val (x, msg') = resume(())
    (x, msg ++ msg')

fun is_admin(user_id: int): <log> bool
  val result = admins.any(fn(x) x == user_id)
  if !result then
    log("Denied " ++ show(user_id) ++ "\n")
  result

fun all_admins(l): <log> bool
  l.map(is_admin).foldl(True, (&&))

fun main()
  val (result, log) = hLog { [1,2,3,4].all(is_admin) }
```

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. This is a mechanism to define higher-order effects by defining a translation into a program with only algebraic effects. This means that evaluation of higher-order effects is a two-step process: first higher-order effects are elaborated into algebraic effects, which are then evaluated. Like handlers, elaborations are modular, and it is possible to define multiple elaborations for a single effect.

Therefore, there currently exist languages with algebraic effects and there is a theory for hefty algebras, but there is no language yet based on hefty algebras. This is the gap in the research that this thesis aims to fill. The question we therefore wish to answer is:

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

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

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

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

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

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

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

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

Write that README.md

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, 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 extend extends the theory of algebraic effects with higher-order effects. How this is achieved is discussed in Chapter 3.

Elaine is based on the theory of hefty algebras, which is an extension of the theory of algebraic effects. Hence, the theory of algebraic effects also applies 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.

2.1 Monads

TODO: Some more background and citations on monads

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

While many descriptions of monads using 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.

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

This class tells us that we can construct a value of `m a` for any type `a` and for any monad `m` using **`return`**. 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 **`Nothing`** is encountered. To represent the intention of the operation, we define an `abort` function which just returns **`Nothing`**.

```haskell
data Maybe a
  = Just a
  | Nothing

class Monad Maybe where
  return = Just

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

abort :: Maybe a
abort = Nothing
```

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 `>>=`.

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

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

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

A more involved example is the `State` monad. If we were to keep track of state manually a function that modifies state would need to take some state 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.

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

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

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

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

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

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

```haskell
instance Monad (State s) where
  return x = State $ \s -> (s, x)

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

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

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

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

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

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

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

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

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

main = print $ runState 0 bar # prints 0 + 5 + 6 + 7 = 18
```

This is the power of monads: they allow us to abstract the effectful operations away, while also signalling the effects that a function requires in the return type. In the final example, we do not have to think about how the `State` monad works anymore, but only use the `get` and `put` operations to build complex computations. With this abstraction, there is a separation between the interface and the implementation of the state effect. This modularity is one of the core motivation of any theory 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

```haskell
multipleIncs = do
  inc 5
  inc 6
  inc 7
```

If the results from the `inc` computations needs to be used, the `<-` operator, which is part of the 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.

```haskell
multipleIncs = do
  a <- inc 5
  b <- inc 6
  c <- inc 7
  return (a + b + c)

-- which is equivalent to
multipleIncs =
  inc 5 >>= \a ->
    inc 6 >>= \b ->
      inc 7 >>= \c ->
        return (a + b + c)
```

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 looks as follows.

```haskell
decrement :: State Int (Maybe Int)
decrement = get >>= \s ->
              if s > 0
              then put (s - 1) >>= get >>= pure . pure
              else pure abort
```

What's 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. If we would then want to change the type to `Maybe (State Int Int)`, the entire computation would need to be rewritten.

```haskell
decrement :: Maybe (State Int Int)
decrement = (pure get) >>= \s ->
              if s > 0
              then pure (put (s - 1) >>= get)
              else abort
```

For complex computations, this quickly gets complicated. Instead, there could be some combined monad `MaybeAndState` that combines the operations of both monads.

```haskell
decrement :: MaybeAndState Int Int
decrement = get >>= \s ->
              if s > 0
              then put (s - 1) >>= get
              else abort
```

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.

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

instance Functor f => Monad (Free f) where
  return = Pure
  Pure x >>= f = f x
  Do g >>= f = Do $ fmap (>>= f) g
```

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.

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

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.

```haskell
data Abort k = Abort
  deriving Functor
```

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.

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

data End k
  deriving Functor
```

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 typeclass relation `<`, which defines an injection from a functor `f` to any composed functor `g` that contains `f`. We can use this injection to define `get`, `put` and `abort`. These convenience functions are called *smart constructors*.

TODO: Parts of this might not be important

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

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

get :: State s < f => Free f s
get = Do $ inj $ Get Pure

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

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

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.

```haskell
decrement :: Free (State Int + Abort + End) Int
decrement = get >>= \s ->
              if s > 0
              then put (s - 1) >>= pure (s - 1)
              else abort
```

However, there is no way to evaluate this computation, because the free monad is just a syntactic representation of the computation. To do that, there needs to be a function with the type

```
Free (f + f') a -> Free f' b
```

for every `f` and finally a `Free End a -> a` to reduce the free monad to a final value. Following Plotkin and Pretnar (2009), these functions are called *handlers*. Using a fold over the free monad, the definition of the handlers can be reduced to 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. This paremeter is used to thread state through the computation. Therefore, `handle` is defined as follows.

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

handle :: (Functor f, Functor f')
       => (a -> p -> Free f' b)
       -> (f (p -> Free f' b) -> p -> Free f' b)
       -> Free (f + f') a -> p -> Free f' b
handle ret f = fold ret $
  \case
    L x -> f x
    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 and finally we reduce the free monad with only the `End` functor to the final value. So, `decrement` requires handlers for state and abort.

```haskell
-- The Do case does not need to be handled since End cannot be
-- constructed
handleEnd :: Free End a -> a
handleEnd (Pure a) = a

handleAbort :: Functor f => Free (Abort + f) a -> Free f (Maybe a)
handleAbort c = handle
  (\a _ -> Pure $ Just a)
  (\Abort () -> Pure Nothing)
  c ()

handleState :: Functor f => s -> Free (State s + f) a -> Free f (s, a)
handleState = flip $ handle
  (\x s -> pure (s, x))
  (\x s -> case x of
      Put s' k -> k s'
      Get k -> k s s)

result :: (Int, Maybe Int)
result = handleEnd $ handleState (0::Int) $ handleAbort decrement
```

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

While the plumbing needed for a free monad is extensive, it is worth considering what it provides. First, we can combine multiple functors in our type signatures. Second, we can define operations that work for any effect composition that contains an effect. Third, we can provide modular handlers that handle a single effect from the composed functors. If all effects are defined in this way, then effect is automatically compatible with all other effects.

Finally, we have not only gained modularity for the effects themselves, but also for the handlers. The effects have become a specification of the syntax, while the handlers provide the semantics. Within this framework, the type and definition of the computation then does not need to be changed, only the handler. There is nothing preventing different implementations of the handlers. It is, for example, possible to define a state handler in which `put` operations are ignored, keeping the state is constant.

2.3 Algebraic Effects

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

The algebraicity property states that the `>>=` operation distributes over the computation parameters of an operation. This means that if there is some operation `op` that has some parameter of type `k` then the following computations should be equivalent:

$$\texttt{(op k) >>= k'} \quad == \quad \texttt{op (k >>= k')}$$

By construction, this holds for any effect we have defined in the previous section. This can be derived from the definitions of `>>=` on `Free` and `fmap` for the effects. For example, we can apply the definitions to `Free (State s)`.

Haskell

```haskell
-- we start with:
put s >>= k
-- expand smart constructor:
Do (Put s (Pure ())) >>= k
-- apply >>= of Free:
Do $ fmap (>>= k) (Put s (Pure ()))
-- apply fmap of Put:
Do $ Put s (Pure () >>= k)
-- simplify
Do $ Put s k
```

The state and abort effects satisfy this property, along with effects for non-determinism, cooperative concurrency. However, *higher-order effects* such as exception catching and the reader effect with a local operation are not algebraic. Those effects are discussed extensively in Chapter 3.

and more

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 thus 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 -> e b`, which is analogous to `a -> 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 -> 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

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

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 `<a,b>` and `<b,a>`.

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

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

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.

```koka
effect abort
  ctl abort(): a

effect state<a>
  ctl get(): a
  ctl put(x: a): ()
```

FEEDBACK: Deze alineas verdienen wat liefde en aandacht.

The equivalent of `Free (State s + Abort + End) a` becomes `<state<s>,abort> a`. The equivalent of a handler would then be a function which takes `() -> <f|e> a` and returns `<|e> 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 is wrapped in another function which takes an initial value for the state.

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

fun hState(init, c)
  fun h(c')
    with handler
      return(x)  fn(s) (s, x)
      ctl get()  fn(s) resume(s)(s)
      ctl put(n) fn(_) resume(())(n)
    c'()

  h(c)(init)
```

TODO: Note that Koka is more “inspired” by free monads, because the arbitrary return types are hard to do in Haskell.

This saves us from specifying some details, but the structure is largely the same as with the free monad encoding. The larger differences become apparent when we want to use the effects. A port of the decrement function is listed below.

```koka
fun decrement(): <state<int>,abort> int
  val s = get()
  if s == 0 then
    abort()

  put(s - 1)
  s - 1

fun printMaybe(m: maybe<int>)
  match m
    Just(x) -> println(x)
    Nothing -> println("nothing!")

fun main()
  printMaybe(hAbort { hState(3, foo) } ) // prints "2"
  printMaybe(hAbort { hState(0, foo) } ) // prints "nothing!"
```

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

```koka
put(get() + 1)
```

```haskell
do
  x <- get
  put (x + 1)
```

In Koka, effectful operations can be used anywhere as long as they are wrapped in a corresponding handler. In the end, the syntax is closer to imperative programming languages than functional programming languages.

However, the type system still very much resembles that of a functional language. This is important because this means that we have not lost any of the type safety that the monadic treatment of effects provides. The signature of a function in Koka still gives a complete specification of all effects that a function might perform. In imperative languages, this information is entirely missing from the function signature. For example, the type system can to assert that a function is entirely pure. In the listing below, the `<>` in the type of the function asserts that it does not require effects, yet the `println` function requires an effect. Hence, Koka's type checker will yield a type error.

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

As will become clear in Chapter 4, Elaine takes a lot of inspiration from Koka. Handlers and effects are defined in the same way, modulo some syntactical difference. What sets Elaine 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 and selects the branch to take based on the boolean. It is essentially an **`if-else`** expression expressed as an effect.

```haskell
data Branch k = Branch Bool k k

branch :: Branch < f => Bool -> Free f a -> Free f a -> Free f a
branch b ifTrue ifFalse :: Do $ inj $ Branch b ifTrue ifFalse
```

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

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

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

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

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 therefore algebraic on its own. 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.

```haskell
data Reader a k = Ask (a -> k) | Local (a -> a) k k

ask       = Do $ inj $ Get Pure
local f c = Do $ inj $ Local f c (Pure ())
```

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

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

Only the first `get` 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, do that yield the following computation.

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

Now, both `get` 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 can, for example, represent an error message. The **`catch`** operation should then evaluate its first parameter and jump to the second if it fails, much like the try-catch constructs of languages with effects.

```haskell
data Except a k = Throw a | Catch k k

throw     = Do $ inj $ Throw
catch a b = Do $ inj $ Catch a b
```

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

```haskell
catch (pure False)  (pure True)  >>= throw -- -> throw False
-- then distributing >>= yields
catch (throw False) (throw True)           -- -> throw True
```

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` with 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.

```haskell
catch c1 c2 =
  case handleThrow c1 of
    Left e -> c2
    Right a -> pure a
```

The distinction between effects which are and which are not algebraic has been described as the difference between *effect constructors* and *effect deconstructors* (Plotkin and Power 2003). The `local` and `catch` operations have to act on effectful computations and change the meaning of the effects in that computation. So, they have to deconstruct the effects in their computations 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 the hefty tree. 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.

```haskell
-- A regular functor
class Functor f where
  fmap :: (a -> b) -> f a -> f b

-- An higher-order functor
class (forall f. Functor (h f)) => HOFunctor h where
  hmap :: (f a -> g a) -> (h f a -> h g a)
```

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

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

-- hefty tree
data Hefty h a
  = Return a
  | Do (h (Hefty h) (Hefty h a))
```

A hefty tree and a free monad are very similar: we can define the `>>=`, `<` 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.

```haskell
data Lift f (m :: * -> *) k = Lift (f k)
  deriving Functor

instance Functor f => HOFunctor (Lift f) where
  hmap _ (Lift x) = Lift x
```

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

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

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:

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

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.

```haskell
hfold :: HOFunctor h
      => (forall a. a -> g a)
      -> (forall a. h g (g a) -> g a)
      -> Hefty h a
      -> g a
hfold gen _   (Return x) = gen x
hfold gen alg (Do x)     =
  ha alg (fmap (hfold gen alg) (hmap (hfold gen alg) x))

elab :: HOFunctor h
     => (forall a. h (Free f) (Free f a) -> Free f a)
     -> Hefty h a
     -> Free f a
elab elabs = hfold Pure elabs
```

[1]We 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 `:<`.

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

```haskell
elabLift :: g < f => Lift g (Free f) (Free f a) -> Free f a
elabLift (Lift x) = Op (inj x)
```

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

```haskell
data Reader r k = Local r k k

elabReader :: Ask r < f
           => Reader r (Free f) (Free f a)
           -> Free f a
elabReader (Local f m k) = ask >>= \r -> handle (hAsk (f r)) m >>= k
```

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

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

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

```haskell
handleA :: Free (f + f') a
        -> Free f' a

elabH   :: Comp (h + h') f a
        -> Comp h' f a
```

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

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

Otherwise, Elaine follows the framework of hefty algebras closely. This concludes the theory that Elaine relies on. Without further ado then, we will dive into the design of Elaine 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 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 types with s 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. Second, 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!"`.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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 and capitalize types.

The primitive types are `String`, `Bool` and `Int` for strings, booleans and integers respectively. The types for let bindings, function arguments and return types can be explicitly specified.

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

let triple = fn(x: Int) Int { mul(3, x) };
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.

```Elaine
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.

```Elaine
let f = fn(x: a) (a, Int) {
    (x, 5)
};
let y = f("hello");
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 under the `maybe` module.

```
Elaine
use std;

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

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

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

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

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

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

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

4.5 Recursion & Loops

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

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

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

```
use std;

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

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

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

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

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

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

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

4.6 Algebraic Effects

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

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

Figure 4.2 lists examples of effect declarations for the `Abort`, `Ask`, `State` and `Write` effects. We will refer to those declarations throughout this section. For the listings in this section, one can assume that these declarations are used. The `Abort` effect is meant to exit the computation. `Ask` provides some integer value to the computation, much like a global constant. `State` corresponds to the `State` monad in Haskell. Finally, `Write` allows us to write some string value somewhere. We will be using this to provide a substitute for writing to standard output.

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

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

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

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

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 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 `resume`s execution with some values. The `resume` function represents the continuation of the program after the operation. The simplest handler for `Ask` we can write is one which yields some constant value.

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

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

Of course, it would be cumbersome to write a separate handler for every value we would like to provide. Since handlers are first-class values, we can return the handler from a function to simplify the code. This pattern is quite common to create dynamic handlers with small variations.

```
Elaine
let hAsk = fn(v: Int) {
    handler { ask() { resume(v) } }
};

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

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

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

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

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

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

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

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

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

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

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

```

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

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

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

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

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

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

4.6.2 Effect Rows

All types in Elaine have an effect row. So far, we have omitted the effect rows, because effect rows can be inferred by the type checker. Effect rows represent the set of effects that need to be handled to obtain the value in a computation. For simple values, that effect row is empty, denoted `<>`. For example, an integer has type `<> Int`. With explicit effect row, the `square` function in the previous section could therefore have been written as below.

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

Table 4.1: Examples of effect row unification.

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

Simple effect rows consist of a list of effect names separated by commas. The return type of a function that returns an integer and uses the `Ask` and `State` effects has type `<Ask,State> Int` or, equivalently `<State,Ask> Int`. The order of effects in effect rows is irrelevant. However, the multiplicity is important, that is, the effect rows `<State,State>` and `<State>` are not equivalent. To capture the equivalence between effect rows, we therefore model them as multisets.

Additionally, we can extend effect rows with other effect rows. 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*.

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

$$\texttt{<A1,...,AN|e>} = \texttt{<B1,...,BM|f>},$$

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.

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.

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

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

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

4.7 Functions Generic over Effects

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.

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

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

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

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

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

```
Elaine
let rec while = fn(
    predicate: fn() <|e> Bool,
    body: fn() <|e> ()
) <|e> (){
    if predicate() {
        ()
    } else {
        body()
        while(predicate, body)
    }
};
```

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` operations is written `catch!`.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

effect Throw {
    throw(String) a
}

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

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

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

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

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

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

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

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

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

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

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

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

Chapter 5

Implicit Elaboration Resolution

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.

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

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

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

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

use local;

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

However, while useful, this feature only saves a few characters in the examples above. It becomes more important when multiple higher-order effects are involved: and **`elab`** without argument will elaborate all higher-order effects in the sub-computation. For instance, if elaborations for both `Exception` and `Reader` are in scope, the following program works.

```
Elaine
let main = handle[hAsk(2)] handle[hThrow] elab {
    local!(double, {
        if gt(ask(), 3) {
            throw()
        } else {
            add(ask(), 4)
        }
    })
}
```

This relies on the fact that the order in which elaboration are applied does not affect the semantics of the program as explained in Section 3.4. To make the inference predictable, we require that an implicit elaboration must elaborate all higher-order effects in the sub-computation.

A problem with this feature arises when multiple elaborations for a single effect are in scope; which one should then be used? To keep the result of the inference predictable and deterministic, the type checker should yield a type error in this case. Hence, if type checking succeeds, then the inference procedure has found exactly one elaboration to apply for each higher-order effect. If not, the elaboration cannot be inferred and must be written explicitly.

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

let main = elab { local!(double, ask!()) }; # Type error here!
```

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

To infer the elaborations, the type checker first analyses the sub-expression. This will yield some computation type with an effect row containing both higher-order and algebraic effects: `<H1!, ..., HN!, A1, ..., AM>`. It then checks the type environment to look for elaborations `e1`, ..., `eN` which elaborate `H1!`, ..., `HN!`, respectively. Only elaborations that are directly in scope are considered, so if an elaboration resides in another module, it needs be imported first. For each higher-order effect, there must be exactly one elaboration.

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

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 `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 it is defined as below.

```haskell
newtype Identity a = Identity a
```

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`

```haskell
instance MonadState s (StateT s m) where
  -- definitions omitted

instance MonadState s m => MonadState s (WriteT m) where
  -- definitions omitted
```

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

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

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 an informal and imperfect comparison, since scoped effects and hefty algebras are evaluated in very different ways.

Elaine
```
effect ScopedEffect! {
    scoped_operation!(a) a
}

let eScoped = elaboration ScopedEffect! -> AlgebraicEffect {
    scoped_operation!(a) {
        handle[endoAlg] a
    }
};
```

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 Eff (Bauer and Pretnar 2015), Koka (Leijen 2014), OCaml citation needed, and Frank (Lindley, McBride, and McLaughlin 2017). In all of these languages, effect row variables can be used to abstract over effects. For example, the signature of the `map` function is in Koka is given below and is similar to the signature of `map` in Elaine.

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

Other languages choose a more implicit syntax for effect polymorphism. Frank (Lindley, McBride, and McLaughlin 2017) opts to have the empty effect row represent the *ambient effects*. 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

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

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) for better performance. 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/rosefromthedead/effing-mad`

Chapter 7

Conclusion

The study 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.

The examples throughout this thesis and in Appendix A also motivate why support for higher-order effects can a useful, since we can easily define modular operations than 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.

Bibliography

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 09/09/2023).

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

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

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

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

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

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

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).

Hutton, Graham and Erik Meijer (1996). *Monadic parser combinators.* URL: `https://nottingham-repository.worktribe.com/output/1024440`.

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: 10.1145/2887747.2804319. URL: https://dl.acm.org/doi/10.1145/2887747.2804319 (visited on 06/03/2023).

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

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

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

— (Mar. 15, 2023). *The Koka Programming Language*. URL: https://koka-lang.github.io/koka/doc/book.html (visited on 06/13/2023).

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

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 ’14: High Integrity Language Technology ACM SIGAda Annual Conference. Portland Oregon USA: ACM, pp. 103–104. ISBN: 978-1-4503-3217-0. DOI: 10.1145/2663171.2663188. URL: https://dl.acm.org/doi/10.1145/2663171.2663188 (visited on 09/16/2023).

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: 10.1016/0022-0000(78)90014-4. URL: https://linkinghub.elsevier.com/retrieve/pii/0022000078900144 (visited on 09/16/2023).

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

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

Piróg, Maciej et al. (July 9, 2018). “Syntax and Semantics for Operations with Scopes”. In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS ’18: 33rd Annual ACM/IEEE Symposium on Logic in Computer Science. Oxford 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 09/11/2023).

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

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

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

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).

Appendix A

Elaine Example Programs

This chapter contains longer Elaine samples with some additional explanation.

A.1 A naive SAT solver

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

```
Elaine
use std;

effect Yield {
   yield(String) ()
}

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

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

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

```

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

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

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

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

A.2 The Reader Effect

TODO: explain

```
Elaine
use std;

effect Ask {
    ask() Int
}

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

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

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

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

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

A.3 Structured Logging

TODO: explain

```
Elaine
use std;

effect Write {
    write(String) ()
}

effect Read {
    ask() String
}

effect Log! {
    context!(String, a) a
    log!(String) ()
}

let hRead = fn(v: String) {
    handler {
        return(x) { x }
        ask() { resume(v) }
    }
};

let hWrite = handler {
    return(x) { "" }
    write(m) {
        let rest = resume(());
        let msg = concat(m, "\n");
        concat(msg, rest)
    }
};

let eLog = elaboration Log! -> <Read,Write> {
    context!(s, c) {
        let new_context = concat(concat(ask(), s), ":");
        handle[hRead(new_context)] c
    }
    log!(m) {
        write(concat(concat(ask(), " "), m))
    }
};

let main = handle[hRead("")] handle[hWrite] elab[eLog] {
    context!("main", {
        log!("msg1");
        context!("foo", {
            log!("msg2")
        });
        context!("bar", {
            log!("msg3")
        })
    })
};
```

A.4 Parser Combinators

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

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

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

Elaine

```
use std;
use maybe;
use list;
use state_str;
use abort;

effect Parse! {
    # Signal that this branch has failed to parse
    fail!() a
    # Try to apply the parser, reset the state if it fails
    try!(a) Maybe[a]
    # Eat the string from the start, which may fail if the string is not the prefix o
    eat!(String) String
}

let eParse = elaboration Parse! -> <State,Abort> {
    fail!() { abort() }
    try!(x) {
        let old_state = get();
        match handle[hAbort] x {
            Just(res) => Just(res),
```

[1]`https://github.com/mrkkrp/megaparsec`

```
                Nothing() => {
                    put(old_state);
                    Nothing()
                }
            }
        }
        eat!(s) {
            let state = get();
            if is_prefix(s, state) {
                let new_state = drop(length(s), state);
                put(new_state);
                s
            } else {
                abort()
            }
        }
};

### Combinators
let alt2 = fn(a, b) {
    match try!(a()) {
        Just(x) => x,
        Nothing() => b(),
    }
};

let rec alt = fn(parsers) {
    match parsers {
        Cons(p, ps) => alt2(p, fn() { alt(ps) }),
        Nil() => fail!(),
    }
};

let rec many = fn(p) {
    match try!(p()) {
        Just(x) => Cons(x, many(p)),
        Nothing() => Nil(),
    }
};

let separated = fn(p: fn() <Parse!> a, separator: fn() <Parse!> b) <Parse!> List[a
    match try!(p()) {
        Just(x) => Cons(x, many(fn() {separator(); p()})),
        Nothing() => Nil()
    }
};

### Parsers
# Parse a token specified as a string
let token = fn(s) { eat!(s) };

let one_of = fn(s) {
    fn() {
        alt(map( fn(x) { fn() { token(x) } }, explode(s)))
    }
};

# Parse a single digit
let digit = one_of("0123456789");
let str_char = one_of("0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
let white_one = one_of(" \n\t");
let white = fn() { many(white_one); () };

let tokenws = fn(s) {
    let t = token(s);
    white();
    t
};

let comma_separated = fn(p: fn() <Parse!> a) <Parse!> List[a] {
    separated(p, fn() { tokenws(",") })
};

# Parse as many digits as possible
let number = fn() { join(many(digit)) };

type Json {
    JsonString(String),
    JsonInt(String),
    JsonArray(List[Json]),
    JsonObject(List[(String, Json)]),
}

let string = fn() <Parse!> String {
    token("\"");
    let s = join(many(str_char));
    tokenws("\"");
    s
};

let key_value = fn(value: fn() <Parse!> Json) <Parse!> (String, Json) {
    let k = string();
    tokenws(":");
    (k, value())
};

let object = fn(value: fn() <Parse!> Json) <Parse!> Json {
    tokenws("{");
    let kvs = comma_separated(fn() { key_value(value) });
    tokenws("}");
    JsonObject(kvs)
};
```

```

let array = fn(value) {
    tokenws("[");
    let values = comma_separated(value);
    tokenws("]");
    JsonArray(values)
};

let rec value = fn() {
    alt([
        fn() { array(value) },
        fn() { object(value) },
        fn() { JsonString(string()) },
        fn() { JsonInt(number()) },
    ])
};

let parse = fn(parser, input) {
    let f = handle[hState] handle[hAbort] elab[eParse] parser();
    f(input)
};

let main = parse(
    value,
    "{\"key1\": 123, \"key2\": [1,2,3], \"key3\": \"some string\"}"
);
```

Appendix B

Elaine Specification

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

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

B.1 Syntax definition

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

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

The full syntax definition is given in Figure B.1. For convenience, we define and use several extensions to BNF:

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

B.2 Effect row semantics

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

One possible treatment is then to model effect rows as sets. However, as noted by Leijen (2014), this leads to some problems. Consider the following (abridged) program.

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

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

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

Figure B.1: Syntax definition of Elaine

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

The treatment of effect rows then simplifies if duplicated effects are allowed (Leijen 2014). Hence, we use multisets to model effect rows, meaning that the row $\langle A, B, B, C\rangle$ is represented by the multiset $\{A, B, B, C\}$. This yields a semantics where the multiplicity of effects is significant, but the order is not.

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

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

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

We define the operation set as follows:

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

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

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

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

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

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

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

TODO: Talk about (Leijen 2005, 2014).

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

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

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

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

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

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

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

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

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

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

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

B.3 Typing judgments

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

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

INFO: A Γ_T for data types might be added.

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

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

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

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

B.3.1 Type inference

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

QUESTION: Koka requires an empty effect row. Why?

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

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

B.3.2 Expressions

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

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

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

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

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

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

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.

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

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

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

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

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

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.

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

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

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

B.3.5 Higher-Order Effects and Elaborations

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

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

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

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.

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

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

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

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

B.4 Desugaring

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

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

B.5 Semantics

The semantics of Elaine are defined as reduction semantics.

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

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

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

TODO: Add some explanation

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

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

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

B.6 Standard Library

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

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

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

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

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