Chapter 1

Freer-Monadic Effects

1.1 Interleaved Effects

To *interleave* effects is to use multiple effects at the level. For example, abstracting away from a particular effects implementation, the following code uses three effects at the same level:

```
1 term get-username (_:unit) : string
2   := do
3         { name ← (get input from the user)
4         ; (set a variable username to the value of name)
5         ; (if username is "Henry", then throw an exception;
6            otherwise result in username) }
```

In section **??**, we reflected on this problem as the problem of *composing monads* in $\mathbb{C}$. For $\mathbb{C}$, the code written above would have to be significantly modified in order to work. The performance on line 3 has type `io string`, the performance on line 4 has type `mutable string unit`, and the performance on line 5 has type `exceptional string` — so they cannot be directly sequenced together as the same level. In order to be sequenced, each term must be of the same monad.[1]

In chapter **??**, algebraic effect handlers were introduced as a framework for implementing effects that maintained the generality and strictness of monadic effects but also allowed for interleaving effects. The `get-username` example could be written in $\mathbb{D}$ with the same structure as presented in the example — disparate effects can be sequenced as an example of interleaving. However, algebraic effect handlers sacrificed some of the type-safety of $\mathbb{C}$ and re-introduced a reliance on language-external reduction contexts (for handlers and performances).

Although monadic effects and algebraic effect handlers have been presented thus far as completely orthogonal approaches to implementing, it turns out that there is a way to implement a variant of algebraic effect handlers with $\mathbb{A}$ using monads. The strategy is to

[1] TODO: mention monad transformers

define a generalized monad that is parametrized by an effect type as well as a result type.

We will construct a type `Freer : (Type → Type) → Type → Type` which lifts particular effects of type `Type → Type` into a single overarching effect type. For example, `Freer (mutable σ) α` is the lifted mutability effect. Additionally, we shall construct a term `freer (M : Type → Type) (α : Type) : M α → Freer M α` that lifts actions (terms) of `M` to be actions of the overarching effect type. Given these constructions, disparate effects can be intertwined since they will each be wrapped within the same `Freer` type. Call our language that implements and makes use of freer monads 𝔼.

1.2 Freer Monad

[**TODO**] cite okmij article online and paper

[**TODO**] start by describe goals:

1. preserve typing rules
2. don't require redundant implementations of same effect-style for each instance
3. separate performances and handlers
4. reference to Left Kan Extension (LAN) as being inspiration

Freer monads are first majorly described as a basis for a generalized effect framework in **?**. However the paper approaches freer monads as a generalization of *free monads* and *monad transformers*,[2] whereas this work approaches freer monads as a monadic implementation of algebraic effect handlers. The idea behind freer monads is to lift types `υ : Type → Type` to monad instances in a generalized way. Such a lifting is described to yield a monad instance "for free" because no monadic structure is required of `υ`, yet monadic structure involving `υ` is produced.[3] Lifting can be thought of in comparison to instantiating type-classes: the `Monad` type-class requires each instance to individually implement monadic structure; the `Freer` type generally lifts a type to a monadic structure that requires no implementation.

So, how can this be done? Our goal is, given `υ : Type → Type`, to define a type `Freer : (Type → Type) → Type → Type` such that we can instantiate `Monad (Freer υ)` parametrized by result type `α`. Recall the monad capabilities: lifting, mapping, and binding. With these in mind, consider the following definition:

Listing 1.1: Definition of `Freer`

```
type Freer (υ : Type → Type) (α : Type) : Type
```

[2]Describing these structures is beyond the scope of this work.

[3]Similarly the idea behind free monads is to lift `υ` to monad instances in a generalized way that requires `υ` be a functor. So since fre*er* monads do not require `υ` to be a functor, they yield a monad instance "for fre*er*."

```
  := pure   : α → Freer υ α
  | impure : (χ:Type) ⇒ υ χ → (χ → Freer υ α) → Freer υ α.
```

The constructor `pure` clearly corresponds to the lifting monad capability. It is named so to reflect the lifting of a pure α to the impure type `Freer` $\upsilon\ \alpha$[4] The motivation for constructor `impure` is less obvious. It's type signature appears as a sort of mixing between the binding monad capabilities for each of υ on its own and the wrapped `Freer` υ. It is named so to reflect the the handling of ($\upsilon\ \chi$)-terms as effectual, given the (χ → `Freer` $\upsilon\ \alpha$)-continuation, in order to produce a `Freer` $\upsilon\ \alpha$. The intuition is that `impure` $y\ k$ encodes a υ-wrapped χ-term and a continuation k that is waiting for an (unwrapped) χ-term. By itself, a term `impure` $y\ k$ merely encodes such a performance but does not actually *perform* it. It must be provided with a handler that uses y and k to perform the encoded performance and produce the next step of the computation (i.e. a term of type `Freer` $\upsilon\ \alpha$). Such a handler must depend on the structure of the base type υ, and so much be implemented separately for each υ. A **freer-monadic handler**, or just **handler**, has a type of the form `Freer` $\upsilon\ \alpha \to \omega\ \alpha$, where ω is the type of affected results (parametrized by the result type α). This abstracting away of the handling from the performing mimics algebraic effects handlers, in contrast to how monadic effects require the effect handling to be implemented by each effect's monad instance.

To instantiate `Freer` υ as a monad, the implementations of mapping and binding must be provided:

- For mapping f over
  - `pure` a, simply apply f to a.
  - `impure` $y\ k$, maintain y and compose `map` f with k as the new continuation. Note that this case defines `map` recursively.
- For binding in fm the result of
  - `pure` a, simply appled fm to a.
  - `impure` $y\ k$, maintain y and as the new continuation, parametrized by a, bind $k\ a$ to the parameter of fm. Note that this case defines `bind` recursively.

(Observe that the recursive cases will terminate for terms of the form `impure` $y\ k$ as long as they "end" with a pure term i.e. the body of k eventually is a term of the form `pure` a. Termination is not guaranteed otherwise.) The following implements in code the above informal descriptions.

Listing 1.2: Monad instance of `Freer`

[4] The `α` need not necessarily be pure, but it *may* be. Additionally, if `α` is of the form `Free υ β` for the same `υ` and some `β`, then the `join` function (see Appendix A, Prelude for ℂ) can transform the resulting term of type `Free υ (Free υ β)` into a term of type `Free υ β`, joining the nesting of the same monad.

```
instance (υ : Type → Type) ⇒ Monad (Freer υ)
  { lift   a  := pure a
  ; map   f m  := cases m
                  { pure    a   ⇒ pure (f a)
                  | impure y k ⇒ impure y (map f ∘ k) }
  ; bind m fm := cases m
                  { pure    a   ⇒ fm a
                  | impure y k ⇒ impure y (a ⇒ bind (k a) fm) } }
```

Indeed this results in a monad instance for any υ.

So far we have defined the `Freer` type as a way of lifting a type υ `: Type → Type` to a monad instance `Freer` υ (parametrized by result type α). However, we still need a way of lifting a corresponding term y `:` $\upsilon\ \alpha$ to a computation of `Freer` $\upsilon\ \alpha$. Such a lift is simply to wrap y as the first parameter of `impure`, and then provide `pure` as the trivial continuation. The following function implements the informal description in code.

```
term freer (y : υ α) : Freer υ α := impure y pure.
```

1.3 Freer-Monadic Effects

So now that we have described the abstract workings of freer monads, how can they be concretized as particular effects? Going forward, we shall call a type of the form `Freer` $\upsilon\ \alpha$ the **effect type** of the effect structured by the **base type** υ. We shall call a term of type `Freer` $\upsilon\ \alpha$ a freer-υ-computation with α-result. Suppose we have our usual type for the mutability effect: $\sigma \to \sigma \times \alpha$. This will serve a the base type for the freer mutability effect, named simply `mutable` since it is taking the role of the mutability effect.

```
type mutable-base (σ α : Type) : Type := σ → σ × α.
type mutable      (σ α : Type) : Type := Freer (mutable-base σ) α.
```

In order to define the actions `get` and `set` for `mutable`, we can lift the usual monadic-effect implementation via `freer` like so:

```
type get-base (σ:Type) (_:unit) : mutable-base σ σ := s ⇒ (s, s).
type get      (σ:Type) (_:unit) : mutable      σ σ := freer (get-base •).

term set-base (σ α : Type) (s:σ) : mutable-base σ unit := _ ⇒ (s, •).
term set      (σ α : Type) (s:σ) : mutable      σ unit := freer
                                                          (set-base s).
```

In this way, we have constructed a new monadic encoding of the mutability effect without needing to newly instantiate it as a monad! All we needed to do was give the base type and then construct the effect actions as they operate on the base type.

There is clearly a lot of boilerplate structure here that could be simplified — the names `mutable-base`, `get-base`, and `set-base` are only used once to bootstrap their freer-lifts, and the structure of these liftings if very regular. So, we can posit the following notation to prune the process down to a standard pattern for defining freer-monadic effects.

```
effect ⟦«type-param»:«kind»⟧_e «effect-name» lifts «type»_*
  { ⟦ «action-name»_i ⟦«type-param»:«kind»⟧_i ⟦«term-param»:«type»⟧_i
      : «type»_* «type»_i
      := «term»_i ; ⟧ }

::=

type «effect-name» ⟦«type-param»:«kind»⟧_e := Freer «type»_*.

⟦ term «action-name»_i ⟦«type-param»:«kind»⟧_e ⟦«type-param»:«kind»⟧_i
          ⟦«term-param»:«type»⟧_i
    : «effect-name» «type»_i
    := freer «term»_i. ⟧
```

Using this notation, the definition of the freer-monadic mutability effect is written as the following:

Listing 1.3: Definitions for the mutability effect

```
effect (σ:Type) ⇒ mutable σ
 lifts (α:Type) ⇒ σ → σ × α
  { get (_:unit) := s ⇒ (s, s)
  ; set (s:σ)    := _ ⇒ (s, •) }.
```

Finally, we can provide a `initialize : mutable` σ α `→` σ `→` $\sigma \times \alpha$ function that acts as a handler of the mutability effect.

```
term initialize (σ α : Type) (m : mutable σ α) (s:σ) : σ × α
  := cases m
      { pure   a   ⇒ (s, a)
      | impure y k ⇒ let (s', a) := y s in initialize (k a) s' }.
```

While mutability has one canonical handler, later examples will demonstrate effects that could have several different handlers.

[**TODO**] mutability example from prev chapters

1.4 Examples of Freer-Monadic Effects

1.4.1 Example: Exception

The exception effect has base type $\varepsilon \oplus \alpha$, where ε is the exception type and α is the valid type. The usual exception actions produce the left or right construction of the exception type. We can define the freer-monadic exception effect as follows:

```
effect (ε:Type) ⇒ exceptional ε
 lifts (α:Type) ⇒ ε ⊕ α
  { throw (e:ε) := left e
  ; valid (a:α) := right a }.
```

We shall consider two handlers for the exception effect: `optionalized` and `catching`.

The handler `optionalized` handles an exceptional computation by producing a term of type `ε ⊕ α` encoding the monadic-effects representation of exception.

- For a `pure` a term, treat a as a valid.
- For an `impure (left` e`)` k term, ignore k since the computation has already reached an exceptional, and propogate the thrown e.
- For an `impure (right` a`)` k term, pass a to the continuation k.

The following implements in code the above informal descriptions.

```
term optionalized (ε α : Type) (m : exceptional ε α) : ε ⊕ α
  := cases m
       { pure    a    ⇒ right a
       | impure y k ⇒ cases y
                          { left  e ⇒ left e
                          | right a ⇒ k a }.
```

The handler `catching` handles a exceptional computation, given an exception-continuation $f : \varepsilon \to \alpha$, by producing a term of type α. The α-result is either the valid result of the computation if it is valid, or the exception-continuation applied to the thrown exceptional value.

- For a `pure` a term, treat a as valid.

- For an `impure (left e) k` term, ignore `k` since the computation has already reached an exception, and result in the exception-continuation applied to `e`.
- For an `impure (right a) k` term, pass `a` to the continuation `k`.

The following implements in code the above informal descriptions.

```
term catching (ε α : Type) (f : ε → α) (m : exceptional ε α) : α
  := cases m
      { pure    a    ⇒ right a
      | impure y k ⇒ cases y
                        { left   e ⇒ left (f e)
                        | right a ⇒ k a } }.
```

Recall the safe division function, which is written in 𝔼 as follows:

```
term division (i j : integer)
  : exceptional integer integer
  := if j == 0
      then throw i
      else valid (i/j)
```

We can use the handler `optionalized` to encode the result of a division as either the left of a sum (encoding the numerator) if a division-by-0 was attempted, or the right of a sum (encoding the quotiient) if division was valid.

Listing 1.4: Handling `division` with `optionalized`

```
optionalized (division 4 0)
↠
optionalized (throw 4)
↠
optionalized (freer (left 4))
↠
optionalized (impure (left 4) pure)
↠
optionalized 4
```

We can use the handler `catching` to provide an exception-continuation that triggers for exceptional results. The following example uses the exception-continuation `i ⇒ 0` to effectively provide a default value `0` for exceptional divisions.

Listing 1.5: Handling `division` with `catching`

```
catching (i ⇒ 0) (division 4 0)
↠
catching (i ⇒ 0) (throw 4)
```

```
↠
catching (i ⇒ 0) (freer (left 4))
↠
catching (i ⇒ 0) (impure (left 4) pure)
↠
(i ⇒ 0) 4
↠
0
```

1.4.2 Example: Nondeterminism

The nondeterministism effect has base type `list α`. The usual nondeterministic action samples an element of a list. So we can define the freer-monadic nondeterministism effect as follows:

```
effect nondeterministic
 lifts list
  { sample as := as }.
```

We shall consider one canonical handler for this effect: `possibilities`. This handler gathers all the possible results of a computation, where the performances of `sample` proliferate the computational pathways.

```
term all-possibilities (α : Type) (m : nondeterministic α) : list α
  := cases m
       { pure    a    ⇒ [a]
       | impure y k ⇒
           let next-possibilities     := all-possibilities ∘ k in
           let all-next-possibilities := map next-possibilities y in
           concat all-next-possibilities }

// simpler: concat (map (possibilities ∘ k) y)
```

(For the construction of `map` in the `Monad` instance for `list`, see Appendix A, Prelude for 𝔸). Recall the coin-flipping experiments from chapters ?? and ??. We can represent the sample list of coin-flipping as `[true, false]`, as in ??. The experiment is written in 𝔼 as follows:

```
term coin-flip : nondeterministic boolean
  := sample [true, false].
```

```
term experiment : nondeterministic boolean
  := do
       { a ← coin-flip
       ; b ← coin-flip
       ; lift (a ∧ b) }.
```

Then we can use the handler `possibilities` to gather all the possible results of `experiment`.

```
possibilities experiment
↠
all-possibilities
  (sample [true, false] >>= (a ⇒
    sample [true, false] >>= (b ⇒
      lift (a ∧ b))))
↠ (Definition of sample)
all-possibilities
  (impure [true, false] pure >>= (a ⇒
    impure [true, false] pure >>= (b ⇒
      pure (a ∧ b))))
↠ (Apply bind)
all-possibilities
  (impure [true, false] (c ⇒
    pure c >>= (a ⇒
      impure [true, false] pure >>= (b ⇒
        pure (a ∧ b)))))
↠ (Apply bind)
all-possibilities
  (impure [true, false] (c ⇒
    impure [true, false] pure >>= (b ⇒
      pure (c ∧ b))))
↠ (Apply bind)
all-possibilities
  (impure [true, false] (c ⇒
    impure [true, false] (d ⇒
      pure d >>= (b ⇒
        pure (c ∧ b)))))
↠ (Apply bind)
all-possibilities
  (impure [true, false] (c ⇒
    impure [true, false] (d ⇒
      pure (c ∧ d))))
↠ (Apply all-possibilities)
concat
  (map (all-possibilities ∘ (c ⇒
```

```
          impure [true, false] (d ⇒
            pure (c ∧ d))))
    [true, false])
↠ (Simplify)
concat
  [ all-possibilities
      (impure [true, false] (d ⇒
        pure (true ∧ d)))
  , all-possibilities
      (impure [true, false] (d ⇒
        pure (false ∧ d))) ]
↠ (Apply (x2) all-possibilities)
concat
  [ concat
      [ all-possibilities (pure (true  ∧ true))
      , all-possibilities (pure (true  ∧ false)) ]
  , concat
      [ all-possibilities (pure (false ∧ true))
      , all-possibilities (pure (false ∧ false)) ] ]
↠ (Apply (x4) all-possibilities)
concat
  [ concat [[true  ∧ true] , [true  ∧ false]]
  , concat [[false ∧ true] , [false ∧ false]] ]
↠ (Simplify)
[true ∧ true, true ∧ false, false ∧ true, false ∧ false]
↠ (Simplify)
[true, false, false, false]
```

1.4.3 Example: I/O

For the I/O effect, 𝔻 presented two handlers: one external and one internal. We can mimic this freer-monadically by specifying a single type class `IO` for the I/O effect, and then creating two freer-monadic effects that will each be instantiated of the the `IO` class. The `IO` class specifies the actions required by the I/O effect. Then, the `io` effect lifts instances of `IO` to monad instances. In other words, it creates an effect for each instance of `IO`.

```
class IO (ιo : Type → Type)
  { input-class  : unit → ιo string
  ; output-class : string → ιo unit }.

effect (ιo : Type → Type) {IO ιo} ⇒ io ιo
 lifts ιo
```

```
  { input  _ := input-class •
  ; output s := output-class s }.
```

1.4.3.0.1 External I/O. We require two new primitive introductions: a wrapping type `external`, and its instance of `IO`.

```
primitive type external : Type → Type.

primitive instance IO external.
```

Additionally, the following primitive actually handle the performances of an `io external` term.

```
primitive term run-external : io external α → external α.
```

1.4.3.0.2 Internal I/O. Constructing this handler is more complicated since it requires specific implementation of the effects.

```
type internal (α:Type) : Type
  := list string × list string → list string × list string × α

instance IO internal
  { input  _ := (is, os) ⇒ (tail is, os, head is)
  ; output s := (is, os) ⇒ (is, os ◇ [x], •) }.
```

Additionally we can construct a term `run-internal` that explicitly handles the performance of an internal-I/O effect.

```
term run-internal (is : list string) (os : list string)
         (m : internal-io α)
  : list string × α
  := cases m
      { pure   a    ⇒ ([], a)
      | impure b k ⇒ let (is', os, a) := b is os in
                       run-internal is' os (k a) }.
```

Finally, consider the following experiment. It can be parsed as either an internal I/O effect or an external I/O effect, and handled accordingly.

```
term main (ɩo : Type → Type) {IO ɩo} : io ɩo unit
  := do
      { name ← input •
      ; output ("Hello, " ++ name) }.

term external-main : external unit := main external.

term internal-main : list string × unit
  := run-internal ["Henry", "Blanchette"] [] main.
```

1.5 Poly-Freer Monad

[**TODO**] Explanation of adding a stack of freer monadic effects to large container, which yields another freer monad. this provides composability of effects

[**TODO**] not going to be able to implement this explicitly, so just explain it

[**TODO**] show interleaved effect from beginning

1.6 Discussion

[**TODO**] Discussion of advantages and disadvantages of freer monadic effects

1. fully-typed system for algebraic effect handlers; all the advantages of algebraic effect handlers
2. facilitates generally composable effects
3. not implementable in non-dependently typed languages like Haskell, OCaml, SML, etc. since it requires extensive type-level manipulation
4. potentially very user-unfriendly, incurring many of the original problems with monadic effects