Chapter 1

Algebraic Effect Handlers

1.1 Introduction to Algebraic Effect Handlers

In chapter **??**, we considered language $\mathbb{B}$ which implemented effects by introducing specific language features for each kind of effect. In chapter **??**, we considered language $\mathbb{C}$ which alternatively implemented effects using monads, which as a single language feature, provided a general framework for introducing all new effects. In doing so, monads also added a new layer of complexity, including requirements for: explicit *lifting* of relatively pure values and special binding for using the results of computations. In particular they make it difficult to write code where multiple effects are used at together i.e. composing monadic effects. We would like an extension of $\mathbb{A}$ that provides composable effects while also maintaining the useful features of $\mathbb{C}$.

[**TODO**] detail the maintained features from $\mathbb{C}$

One conceptual step in this direction is the idea of *algebraic effect handlers*. Recall the breakthrough of monadic effects — the implicit context and explicit context of effects could be modeled as language structures (as monads) rather than deferred to reduction. The abstract strategy was to take something intrinsic about the nature of effects in general, and represent them explicitly in a programming language. As another instance of this strategy, observe that there is another way to break down effects — between where the effect is *performed* and where the effect is *handled*. In $\mathbb{B}$: effects are performed by using specific primitive values, and are handled during reduction to affect the program state. In $\mathbb{C}$: effects are performed by using monad-relevant values, and are handled as per the definition of the monad. An alternative way to represent these aspects of effects is to include them both as language structures, but not require them to be overlapping as is the case with monads. In other words, to have a structure of performing effects and a separate structure for handling effects.

[**TODO**] Describe history of AEHs. Cite Prauer, Plotkin, etc. Footnote about Robin Milner's communicating sequential processes. Mention CPS. Inspired by history of delimited control, pi calculus, CPS, actor model and process calculus

Algebraic effect handlers provide an effect framework for this kind of organization. More formally, it breaks down effects into two aspects like so:

- **Performance:** Incurs the *performing* of an effect, affecting the implicit program state and resulting in a value.
- **Handler:** Defines the result of an effect performance, parametrized by the *handler*'s clauses. In its definition, a handler abstracts the context relevant to handling particular performance (in the same way that a function abstracts its parameter).

Additionally, this setup requires an interface to the effects that are to be performed and handled. To way to provide this interface, we introduce the following.

- **Resource:** Specifies a collection of primitive effects by their input and output types. These primitive effects it provides are called **actions**.
- **Channel**: Represents a specific instance of a resource, providing the primitive effects specified by the resource but having a unique implicit state.

A resource specifies the typed interface (collection of actions) to an effect, and the channels of that resource are particular instances of the effect (such as how there can be multiple terms of type `mutable` α for fixed α).

1.2 Language 𝔻

Language 𝔻 implements algebraic effect handlers similarly to the scheme presented in **?**.

1.2.1 Syntax for 𝔻

[**TODO**] english

Table 1.1: Syntax for 𝔻

metavariable	**constructor**	**name**
«*declaration*»	`resource` «*resource-name*» ⟦ `(`«*type-param*»`:`«*kind*»`)` ⟧ `{` ⟦ «*action-name*» ⟦ `(`«*type-param*»`:`«*kind*»`)` ⟧ `:` «*type*» `↗` «*type*» `;` ⟧ `}.`	resource definition
	`channel` «*channel-name*» `:` «*type*»`.`	channel instantiation
«*kind*»	`Resource`	resource kind
«*type*»	«*type*» `↗` «*type*»	action type
	«*type*» `↘` «*type*»	handler type
«*term*»	«*channel-name*»`#`«*action-name*» «*term*»	performance
	`handler` «*term-name*» ⟦ `(`«*type-param*»`:`«*kind*»`)` ⟧ `:` «*kind*» `{` ⟦ `#`«*action-name*» «*term-parap*» «*term-param*» `⇒` «*term*» `;` ⟧ `}`	handler
	«*term*» `with` «*term*»	handle performance

1.2.2 Primitives for 𝔻

[**TODO**] english

1.2.2.1 Action

The action type is the type of atomic effects provided by a resource. It has two parameters: firstly the input type and secondly the output type. It is necessary for all actions to be represented this way (even if they take a trivial input or result in a trivial output) in order for the to-be-explained framework of performing effects to succeed in generality. The action type serves only as a sort of tag for the signature of the effect it represents — it not have any content. For this reason it is introduced as a new syntactical structure rather than a primitive type.

An action can be thought of a sort of function except that it has no body and the normal $\mathbb{A}$ reduction rules for function applications do not apply to it. An action is a function that *appeals* to a relatively implicit context to dictate its reduction. In this way, the " $\nearrow$ " operator looks similar to the arrow type's arrow, but is tilted upward in appeal to another context.

1.2.2.2 Resource

The kind `Resource` is the kind of *resource types*. A resource type contains a specification for the collection of actions provided by the resource. This specification comes in the form of a collection of *action names* that are each annotated by an action type. These action names are called the actions *provided by* the resource. In general, a declaration

```
resource ρ { g₁ : α₁ ↗ β₁ ; ⋯ ; gₙ : αₙ ↗ βₙ }.
```

declares the type ρ `: Resource` and the terms $g_1 : \alpha_1 \nearrow \beta_1, \ldots, g_n : \alpha_n \nearrow \beta_n$. The typing rule is as follows:

Table 1.2: Typing in $\mathbb{D}$: Resource

$$\textsc{Resource}\quad \frac{\begin{array}{l}(\forall i)\quad \Gamma \vdash \alpha_i : \texttt{Type} \\ (\forall i)\quad \Gamma \vdash \beta_i : \texttt{Type} \\ \texttt{resource}\ \rho\ \{\ g_1 : \alpha_1 \nearrow \beta_1\ ;\ \cdots\ ;\ g_n : \alpha_n \nearrow \beta_n\ \}.\end{array}}{\begin{array}{l}\Gamma \vdash \rho : \texttt{Resource} \\ (\forall i)\quad \Gamma \vdash g_1 : \alpha_1 \nearrow \beta_1\end{array}}$$

For example, consider the following resource declaration:

```
resource Random
  { gen-probability : unit ↗ rational
  ; gen-boolean   : unit ↗ boolean }.
```

This declaration declares the type `Random : Resource` and the terms `gen-probability : unit ↗ rational`, `gen-boolean : unit ↗ boolean`.

1.2.2.3 Channel

A channel is an instance of a resource specification. To declare a channel is to name the new channel and its resource kind.[1] In general, a declaration

```
channel c : ρ.
```

declares the term `c : ρ`, Its typing rule is as follows:

Table 1.3: Typing in 𝔻: Channel

$$\textsc{Channel}\quad \frac{\Gamma \vdash \rho : \texttt{Resource} \qquad \texttt{channel}\ c : \rho.}{c : \rho}$$

For example, consider the following channel declarations:

```
channel random1 : Random.
channel random2 : Random.
```

These declarations declare the terms `random1 : Random` and `random2 : Random`, two different channels for the `Random` resource.

1.2.2.4 Performance

The performance of an action is the use of a channel (for the resource that provides the effect) to invoke the action's effect given a term of the input type. In general, the term

```
c#g a
```

uses channel `c` to perform action `g` with input term `a`. The typing rule for performances is the following:

Table 1.4: Typing in 𝔻: Performance

$$\textsc{Perform}\quad \frac{\Gamma \vdash \rho : \texttt{Resource} \qquad \Gamma \vdash c : \rho \qquad \rho \text{ provides } g \qquad \Gamma \vdash g : \alpha \nearrow \beta \qquad \Gamma \vdash a : \alpha}{\Gamma \vdash c\#g\ a : \beta}$$

For example, consider the following term:

[1] Note thsi special method for introducing channels that seemingly would be accomplished just the same by using the `primitive term` declaration. Using a unique declaration is useful since the declaring of a channel may have effects itself (e.g. declaring a new mutable variable might trigger memory-handling effects) which need to be implemented in an implementation of 𝔻.

```
term random-sum : rational
  := (random1#gen-probability •) + (random2#gen-probability •).
```

Since action `gen-probability` has type `unit ↗ rational`, action `gen-probability` is provided by the resource `Random`, and both of `random1`, `random2`, then each of `(random1#gen-probability •)` and `(random2#gen-probability •)` should result in a rational, which can be added together.

1.2.2.5 Handler

A handler is a term containing an implementation for enacting the effects specified by a certain resource — it is called a handler *for* this resource. Handlers can be used to *handle* (as will be detailed in the section 1.2.2.6) terms that contain such effects, evaluating to a pure result relative to the resource (i.e. no longer has any effects that the resource provides). The implementation clauses that a handler must contain are one for each action provided by the resource, as these actions are the basic units for the resource's effects.

A clause that handles the action `g : χ ↗ υ` has the form `#g x k ⇒ b`, where `x`,`k` are term parameters and `b` is a term. Here, `x` is an input parameter for `g`, `k` is a continuation parametrized by the result of performing `g`, and `b` is a term that encodes the result of performing `g` given input `x` and continuation `k`. More specifically, `k` encodes the rest of the computation to carry out after `g` is performed, with a parameter of type `χ` which appears everywhere the result of performing `g` normally appeared (even if it does not appear anywhere).

Additionally, it is convenient to require two other clauses as well: for *relatively-pure values* (relative to the resource) and the *final result values* (after all effects and values have been handled). A handler can have only one of each of these clauses.

Firstly, a handler's *relatively-pure value clause*, or just **value clause**, encodes a lifting of relatively-pure values to be of the appropriate type reflecting the effect handling (note that this can be the trivial lift, `a ⇒ a`). Such a value clause that handles relatively-pure values has the form `a ⇒ b`, where `a` is a term parameter and `b` is a term. Here, `a` is an input parameter for `b`, and `b` is a term encoding the produced value. For example, an exception handler based on the `optional` data type might lift pure values via the `some` constructor.

Secondly, a handler's *final value clause*, or just **final clause**, encodes some transformation on the result of handling all the other clauses. Such a final clause has the form `b ⇒ c`, where `b` is a term parameter and `c` is a term. Here, `b` is an input parameter for `c`, standing in place of the final result, and `c` is a term encoding some last transformation to apply on `c`. For example, a mutability handler might in its final clause pass an initial value to a continuation parametrized by the state value that is produced through handling the other clauses.

The type of handlers that handle computations with values of type `α` and has result type `β` is `α ↘ β`. The " `↘` " is the the vertical reflection of the action type's operator, indicating that a handler resolves the appeal that an action makes, resulting in a (handled) result; handlers encode the implicit context that actions appeal to and so resolve their appeals.

In general, the term

```
handler
  { #$g_1$ $x_1$ $k_1$ ⇒ $b_1$ ; ⋯ ; $g_n$ $x_n$ $k_n$ ⇒ $b_n$
  ; value $a_v$ ⇒ $b_v$
  ; final $b_f$ ⇒ $c_f$ }
```

combines the action, value, and final clauses mentioned previously. The typing rule is as follows:

Table 1.5: Typing in ⅅ: Handler

$$\textsc{Handler}\quad \frac{\begin{array}{l} \Gamma \vdash \rho : \texttt{Resource} \\ (\forall i)\ \ \rho \text{ provides } g_i \qquad (\forall i)\ \ \Gamma \vdash g_i : \chi_i \nearrow \upsilon_i \\ (\forall i)\ \ \Gamma, x_i:\chi_i, k_i:\upsilon_i \to \beta \vdash b_i:\beta \\ \Gamma \vdash a_v:\alpha \qquad \Gamma \vdash b_v:\beta \\ \Gamma \vdash b_f:\beta \qquad \Gamma \vdash c_f:\gamma \end{array}}{\begin{array}{l} \Gamma \vdash \texttt{handler} \\ \qquad \{\ \#g_1\ x_1\ k_1 \Rightarrow b_1\ ;\ \cdots\ ;\ \#g_n\ x_n\ b_n \Rightarrow y_n \\ \qquad ;\ \texttt{value}\ a_v \Rightarrow b_v \\ \qquad ;\ \texttt{final}\ b_f \Rightarrow c_f\ \} \\ \qquad :\ \rho \to \alpha \searrow \gamma \end{array}}$$

For example, consider the following term:

```
term not-so-randomly (α:Type) : Random → α ↗ α
  := handler
       { #gen-probability _ k ⇒ k 1/2
       ; #gen-boolean     _ k ⇒ k true
       ; value            a   ⇒ a
       ; final            a   ⇒ a }.
```

This handler term handles terms of type α in which `Random`-performances appear, and has result type α. The

1.2.2.6 Handling

So far we have introduced structures for performing effects and defining handlers for these effects. What is still missing is a way to “apply” a handler to a term in way that handles the effects performed in the term via the specification of the handler. This “application” of a handler to a term is called *handling* the term, and the syntactical structure `with` is precisely for handling. In general, the term

```
p with h
```

encodes the handling of the term p (which may contain effect performances) in the way specified by the clauses of the handler h. The intuition behind the syntax is that it encodes "doing" p with h. The actual reduction rules for simplifying this term are given in section 1.2.3.

The typing rule is as follows:

Table 1.6: Typing in $\mathbb{D}$: Handling

$$\textsc{Handling}\quad \frac{\Gamma \vdash p : \alpha \qquad \Gamma \vdash h : \alpha \searrow \beta}{\Gamma \vdash p \ \texttt{with}\ h : \beta}$$

For example, consider the following term (section 1.2.2.7 describes the `do` syntax):

```
term experiment : boolean
  := do
      { b ← random1#gen-boolean •
      ; p ← random1#gen-probability •
      ; b ∧ (1/2 ≤ p) }
    with
      not-so-randomly random1.
```

This term uses the handler `not-so-randomly` to handle the performances using channel `random1` in the computation of the `do` block. Since `b ∧ (1/2 ≤ p)` is a boolean, the declaration that `experiment` has type `boolean` is correct, since `not-so-randomly random1 :` $\alpha \searrow \alpha$ abstracted over all α `: Type`.

1.2.2.7 Sequencing

Listing 1.1: Construction for sequencing

```
term sequence (α β : Type) (_:α) (b:β) : β := b.
```

Listing 1.2: Notations for sequencing

```
(«term»₁:«type»₁) >> («term»₂:«type»₂)
  ::=
    sequence «type»₁ «type»₂ «term»₁ «term»₂

do{ («term»₁:«type»₁) ; ⋯ ; («term»ₙ:«type»ₙ) }
  ::=
    «term»₁ >> ⋯ >> «term»ₙ
```

The operator `>>` is right-associative i.e. `a >> b >> c` associates to `a >> (b >> c)`

We also introduce a matching notation to 𝔹's for binding within `do` blocks:

Listing 1.3: Notation for binding within `do` block

```
do{ ⟦ «term» ; ⟧1 ; «term-param»* ← «term»* ; ⟦ «term» ; ⟧2 }
 ::=
  do{ ⟦ «term» ; ⟧1 ; let «term-param»* := «term»* in (do{ ⟦ «term» ; ⟧2 }) }
```

1.2.3 Reduction Rules for $\mathbb{D}$

1.2.3.0.1 Reduction contexts. The total reduction context Δ is made up of two sub-contexts: $\mathcal{H}$ the handlers context, and $\mathcal{P}$ the performances context. The context $\mathcal{H}$ is a list of handlers, in order from inner-most to outer-most. For example, when considering the term a for reduction within `(`a `with` h_1`) with` h_2, the handlers context would be $\mathcal{H} = [h_2, h_1]$. The context $\mathcal{P}$ is a list of performances, in order from inner-most to outer-most, and from most-recent to least-recent when performances are at the same level. For example, reducing the sequence `do{` a_1 `;` a_2 `;` a_3 `}` would yield the performance context $\mathcal{P} = [a_3, a_2, a_1]$.

1.2.3.0.2 Relative Values. Recall from section **??** that a term is a value if no reduction rules can simplify it. Note though that terms in $\mathbb{B}$ must rely on relatively implicit contexts for reduction — in particular, $\mathcal{H}$. For example: `random1#gen-boolean •` is a value if it appears at the top level, but the same term is not a value if it appears somewhere within the appropriate handling structure, such as `random1#gen-boolean • with not-so-randomly random1`. So, the proposition that a term v is a value, written value v, must be extended to include the reduction context. Thus we introduce the new form "value v relative to $\mathcal{H}$" to abbreviate "v is a value relative to handler context $\mathcal{H}$." The proposition "value v relative to $\mathcal{H}$" is true if value v and there is no handler h among the h such that either of the following is true: v is an action and h has a matching action clause, or h has a `value` clause.

1.2.3.0.3 Performing. The reduction rule Perform dictates how performances interact with the reduction context. This reduction of a performance r`#`g v yields the *pushing* of the performance-representation (r, g, v, a) to the head of the performances context $\mathcal{P}$, where a is a fresh term name that stands in place for the result of the performance. This indicates how (r, g, v, a) is the new inner-most performance, first in priority to get considered for handling. The result of this reduction rule is just a, which will be filled in by whatever the handled result of (r, g, v, a) will be.

Note that the rule Perform has a higher priority than any of the rules for handling. The result of this is that all performances will be enqueued to $\mathcal{P}$ first, and then they will be handled in the order they were enqueued.

1.2.3.0.4 Handling. The reduction rule Handle dictates how to handle a term given a handler. With regards to the reduction contexts, this rule affects both. Reduction of a `with` h yields the adding of h to the inner-most position of $\mathcal{H}$, indicating that h is now the most-prioritized handler. Finally, the result of this reduction is simply a, now to be reduced within the handler context including of h.

1.2.3.0.5 Handling performance. The reduction rule Handle-Performance dictates how to use a given handler h to handle the performance of an action which is highest-priority from $\mathcal{P}$, represented by (r, g, v, a). Let `#`g x k $\Rightarrow$ b be h's clause for handling action g, and w be the value (relative to the handlers in context) being reduced. The clause expects

x to be a value of the input type for action g, which is exactly v since v was given as the argument to performance that included added it to the performance context (as dictated by the rule Perform). Additionally the clause expects k to be a continuation parametrized by the result of the performance, which is exactly $a \Rightarrow w$, since a stands in place of the result of the performance and w is a term referencing a to describe the rest of the computation. So altogether the result of Handle-Performance is b with v passed as x and $a \Rightarrow w$ passed as k, written as the application `(x k ⇒ b) v (a ⇒ w)`.

1.2.3.0.6 Handling value. The reduction rule Handle-Value dictates how to use a given handler h to handle a (relative to h) value v. Let `value a ⇒ b` be the value clause of h. Then the result of Handle-Value should simply be b with v passed as a, written as the application `(a ⇒ b) v`. Additionally, the value clause should only be used once — otherwise, it would apply ad infinitum. So, Handle-Value also removes the value clause from h.

Observe that Handle-Value has a higher priority than Handle-Performance. The result of this is that, once all the performances have been dictated by Perform, rule Handle-Value will apply once, and then the performances will be subsequently handled.

1.2.3.0.7 Handling final. The reduction rule Handle-Final dictates how to use a given handler h to handle a term v that has been so far completely evaluated by applications of Handle-Performance and Handle-Value. Let `final b ⇒ c` be the final clause of h. The result of Handle-Final should simply be c with v passed as b, written as the application `(b ⇒ c) v` (very similar to Handle-Value).

1.2.3.0.8 Raising. The reduction rule Raise dictates that, if a term is a value relative to the inner-most handler, but not a value relative to some next-inner-most handler, then reduction can treat that handler as if it was the inner-most.

[**TODO**] describe how whole structure is working, like, how to conceptualize the overall structure

[**TODO**] define unvalued and unfinaled. also

[**TODO**] $\mathcal{P} \rhd (r\#g\ v, a)$ indicates that $(r\#g\ v, a)$ is being popped off the queue (has heighest priority)

[**TODO**] $(r\#g\ v, a) \rhd \mathcal{P}$ indicates that $(r\#g\ v, a)$ is enqueued into the queue (starts with lowest priority)

Table 1.7: Reduction in $\mathbb{D}$

Simplify
$$\frac{\mathcal{H}\ ;\ \mathcal{P} \parallel a \ \twoheadrightarrow\ \mathcal{H}'\ ;\ \mathcal{P}' \parallel a'}{\mathcal{H}\ ;\ \mathcal{P} \parallel a\ b \ \twoheadrightarrow\ \mathcal{H}'\ ;\ \mathcal{P}' \parallel a'\ b}$$

Simplify
$$\frac{\begin{array}{l}\text{value } v \text{ relative to } \mathcal{H}\\ \mathcal{H}\ ;\ \mathcal{P} \parallel b \ \twoheadrightarrow\ \mathcal{H}'\ ;\ \mathcal{P}' \parallel b'\end{array}}{\mathcal{H}\ ;\ \mathcal{P} \parallel v\ b \ \twoheadrightarrow\ \mathcal{H}'\ ;\ \mathcal{P}' \parallel v\ b'}$$

Sequence
$$\frac{\text{value } v \text{ relative to } \mathcal{H}}{\mathcal{H}\ ;\ \mathcal{P} \parallel v \texttt{ >> } b \ \twoheadrightarrow\ \mathcal{H}\ ;\ \mathcal{P} \parallel b}$$

Bind
$$\frac{\text{value } v \text{ relative to } \mathcal{H}}{\mathcal{H}\ ;\ \mathcal{P} \parallel \texttt{do \{ } x \leftarrow v\ ;\ b \texttt{ \}} \ \twoheadrightarrow\ \mathcal{H}\ ;\ \mathcal{P} \parallel \texttt{do \{ } (x \Rightarrow b)\ v \texttt{ \}}}$$

Raise
$$\frac{\begin{array}{l}\text{value } a \text{ relative to } [h]\\ \mathcal{H}\ ;\ \mathcal{P} \parallel a \ \twoheadrightarrow\ \mathcal{H}'\ ;\ \mathcal{P}' \parallel a'\end{array}}{h :: \mathcal{H}\ ;\ \mathcal{P} \parallel a \ \twoheadrightarrow\ h :: \mathcal{H}'\ ;\ \mathcal{P}' \parallel a'}$$

Handle
$$\mathcal{H}\ ;\ \mathcal{P} \parallel a \texttt{ with } h \ \twoheadrightarrow\ h :: \mathcal{H}\ ;\ \mathcal{P} \parallel a$$

Perform
$$\frac{\text{value } v \text{ relative to } \mathcal{H} \qquad \text{fresh } a}{\mathcal{H}\ ;\ \mathcal{P} \parallel r\#g\ v \ \twoheadrightarrow\ \mathcal{H}\ ;\ (r\#g\ v,\ a) \triangleright \mathcal{P} \parallel a}$$

Handle-Final
$$\frac{\begin{array}{l}\text{value } v \text{ relative to } \mathsf{unfinaled}(h)\\ h := \texttt{handler \{... final } b \Rightarrow c\ ;\ \texttt{...\}}\ r\end{array}}{\mathsf{unvalued}(h) :: \mathcal{H}\ ;\ \mathcal{P} \parallel v \ \twoheadrightarrow\ \mathcal{H}\ ;\ \mathcal{P} \parallel (b \Rightarrow c)\ v}$$

Handle-Value
$$\frac{\begin{array}{l}\text{value } v \text{ relative to } h :: \mathcal{H}\\ h := \texttt{handler \{... value } a \Rightarrow b\ ;\ \texttt{...\}}\ r\end{array}}{h :: \mathcal{H}\ ;\ \mathcal{P} \parallel v \ \twoheadrightarrow\ \mathsf{unvalued}(h) :: \mathcal{H}\ ;\ \mathcal{P} \parallel (a \Rightarrow b)\ v}$$

Handle-Performance
$$\frac{\begin{array}{l}\text{value } w \text{ relative to } h :: \mathcal{H}\\ h := \texttt{handler \{... \#}g\ x\ k \Rightarrow b\ ;\ \texttt{...\}}\ r\end{array}}{\begin{array}{l}h :: \mathcal{H}\ ;\ \mathcal{P} \triangleright (r\#g\ v,\ a) \parallel w \ \twoheadrightarrow\\ h :: \mathcal{H}\ ;\ \mathcal{P} \parallel (x\ k \Rightarrow b)\ v\ (a \Rightarrow w)\end{array}}$$

1.3 Examples

1.3.1 Example: Mutability

1.3.1.1 Resource

The mutability effect can be defined in 𝔻 as a resource as follows, providing its two signature functions here as the resources actions.

Listing 1.4: Resource for mutability.

```
resource Mutable (α : Type)
  { get : unit ↗ α
  ; set : α ↗ unit }.
```

1.3.1.2 Experiment

To test our mutability setup, let us introduce a few channels and impure terms to work with. The following is a selection of channels as instances of the mutability effect with different type arguments. The significance of having these different channels is to specify their type but also to dedicate a unique mutable state to each one, kept track of independently.

Listing 1.5: Channels for mutability

```
// store holds three variables using a product
channel store : Mutable (integer × integer × boolean.).
```

Listing 1.6: Experiment with mutability

```
term experiment : unit
  := do
      { s ← store#get •
      ; x ← part-1 s
      ; y ← part-2 s
      ; store#set (x, y, x < y) }.
```

1.3.1.3 Handling with initialization

[TODO] I should introduce CPS in the introduction to AEHs actually

Handler for a new mutable, given an initial value. The mutable value is handled in `initialize` using another layer of continuation-passing style (CPS). The current continuation `k` has two parameters: the current mutable value and the result of the effect.

Listing 1.7: Handler for mutability.

```
term initialize (α : Type) (a : α) : Mutable α → α ↘ α
  := handler
      { #get   _  k ⇒ (a' ⇒ k a' a')
      ; #set  a' k ⇒ (_ ⇒ k • a')
      ; value a'    ⇒ (s ⇒ a')
      ; final f     ⇒ f a }.
```

To handle the `get` action: the current mutable value is unchanged, and the result is the current mutable value. To handle the `set` action: the current mutable value is updated to a new value, and the result is •. To handle a value: the current mutable value is ignored, and the result is the given value. To handle a final fully-reduced term, which is the form of a function of a current mutable value: the initial mutable value is passed to the term.

```
term h := initialize store (1, 2, false).

[] ; [] ‖ experiment with initialize h

↠ (Handle)
[h] ; []
  ‖ do{ s ← store#get •
      ; x ← part-1 s
      ; y ← part-2 s
      ; store#set (x, y, x < y) }
↠ (Perform)
[h] ; [(store#get •, a1)]
  ‖ do{ s ← a1
      ; x ← part-1 s
      ; y ← part-2 s
      ; store#set (x, y, x < y) }
↠ (Simplify)
[h] ; [(store#get •, a1)]
  ‖ do{ x ← part-1 a1
      ; y ← part-2 a1
      ; store#set (x, y, x < y) }
↠ (Simplify)
[h] ; [(store#get •, a1)]
  ‖ store#set (part-1 a1, part-2 a1, part-1 a1 < part-2 a1)
↠ (Perform)
[h]
; [(store#set (part-1 a1, part-2 a1, part-1 a1 < part-2 a1), a2)
▷ (store#get •, a1)]
  ‖ a2
↠ (Handle-Value)
[unvalued(h)]
; [(store#set (part-1 a1, part-2 a1, part-1 a1 < part-2 a1), a2)
▷ (store#get •, a1)]
  ‖ a ⇒ a2
↠ (Handle-Performance)
[unvalued(h)] ; [(store#get •, a1)]
  ‖ (a k ⇒ (_ ⇒ k • a))
      (store#set (part-1 a1, part-2 a1, part-1 a1 < part-2 a1))
      (a2 ⇒ (a' ⇒ a2))
↠ (Simplify)
[unvalued(h)] ; [(store#get •, a1)]
  ‖ a' ⇒ (part-1 a1, part-2 a1, part-1 a1 < part-2 a1))
```

```
↠ (Handle-Performance)
[unvalued(h)] ; []
  ‖ (_ k ⇒ (a ⇒ k a a))
       •
       (a1 ⇒ (a' ⇒ (part-1 a1, part-2 a1, part-1 a1 < part-2 a1)))
↠ (Simplify)
[unvalued(h)] ; [] ‖ a ⇒ (part-1 a, part-2 a, part-1 a < part-2 a)
↠ (Handle-Final)
[] ; [] ‖ (a ⇒ (part-1 a, part-2 a, part-1 a < part-2 a)) (1, 2, false)
↠ (Simplify)
[] ; [] ‖ (1, 2, 1 < 2)
↠ (Simplify)
[] ; [] ‖ (1, 2, true)
```

1.3.2 Example: Exception

1.3.2.1 Resource

Listing 1.8: Resource for exception

```
resource Exceptional (α : Type)
  { throw : unit ↗ α }.
```

Listing 1.9: Channels for exception

```
channel division-by-0 : Exceptional integer.
channel head-of-nil : Exceptional unit.
```

1.3.2.2 Experiment

[TODO] think of some experiment

Listing 1.10: Experiments with exceptions.

```
term divide-safely (x y : integer) : integer
  := if y != 0 then x/y else division-by-0#throw •.

term head-safely (α : Type) (ls : list α) : α
  := case ls
      { []     ⇒ head-of-nil#throw •
      ; a ∷ _ ⇒ a }.
```

1.3.2.3 Handling exceptions as optionals

Listing 1.11: Handler of exceptions as optionals.

```
term optionalized (α : Type) : Exceptional α → α ↘ optional α
  := handler
      { #throw • _ ⇒ none
      ; value  a   ⇒ some a
      ; final  x   ⇒ x }.
```

Listing 1.12: Handle division safely with optionalization

```
term h := optionalized division-by-0.

[] ; [] ‖ divide-safely 5 0 with h
↠ (Handle)
[h] ; [] ‖ divide-safely 5 0
↠ (Simplify)
[h] ; [] ‖ division-by-0#throw •
↠ (Perform)
[h] ; [(division-by-0#throw •, a)] ‖ a
↠ (Handle-Value)
[unvalued(h)] ; [(division-by-0#throw •, a)] ‖ some a
↠ (Handle-Performance)
[unvalued(h)] ; [] ‖ (x k ⇒ none) • (a ⇒ some a)
↠ (Simplify)
[unvalued(h)] ; [] ‖ none
↠ (Handle-Final)
[] ; [] ‖ none
```

1.3.3 Example: Nondeterminism

1.3.3.1 Resource

[TODO] describe

Listing 1.13: Resource for nondeterministic coin-flipping.

```
// specify a resource for coin-flipping effect
// flip returns true for heads, and false for tails
resource Coin { flip : unit ↗ boolean }.

// create a new Coin channel
channel coin : Coin.
```

1.3.3.2 Experiments

An experiment that counts the number of heads resulting from two `coin#flip`'s.

Listing 1.14: Experiment with nondeterministic coin-flipping.

```
term experiment : boolean
```

```
  := (coin#flip •) ∧ (coin#flip •)
```

1.3.3.3 Handling all possible flips

A handler that accumulates all possible results of the experiment where each flip yields either heads or tails

Listing 1.15: Handler for either heads or tails.

```
term either-heads-or-tails : Coin → integer ↘ list integer
  := handler
      { #flip _ k ⇒ k true ◇ k false
      ; value x   ⇒ [x]
      ; final xs  ⇒ xs }.
```

Listing 1.16: Handle experiment with either heads or tails.

```
term h := either-heads-or-tails coin.

[] ; [] ∥ experiment with h
(Handle) ↠
[h] ; [] ∥ (coin#flip •) ∧ (coin#flip •)
(Perform ×2) ↠
[h] ; [(coin#flip •, a2) ▹ (coin#flip •, a1)] ∥ a2 ∧ a1
(Handle-Value) ↠
[h] ; [(coin#flip •, a2) ▹ (coin#flip •, a1)] ∥ [a2 ∧ a1]
(Handle-Performance ×2)
[h] ; [] ∥ [true ∧ true, false ∧ true, false ∧ true, false ∧ false]
(Simplify) ↠
[h] ; [] ∥ [true, false, false, false]
(Handle-Final) ↠
[] ; [] ∥ [true, false, false, false]
```

1.3.3.4 Handling singular flips

A handler that computes result of experiment where each flip yields heads.

Listing 1.17: Handler for just heads.

```
term just-heads : Coin → integer ↘ integer
  := handler
      { #flip _ k ⇒ k true
      ; value x ⇒ x
      ; final x ⇒ x }.
```

Listing 1.18: Handle experiment with just heads.

```
term h := just-heads coin.

[] ; [] ‖ experiment with just-heads
↠ (Handle)
[h] ; [] ‖ (coin#flip •) ∧ (coin#flip •)
↠ (Perform ×2)
[h] ; [(coin#flip •, a2) ▷ (coin#flip •, a1)] ‖ a2 ∧ a1
↠ (Handle-Value)
[unvalued(h)] ; [(coin#flip •, a2) ▷ (coin#flip •, a1)] ‖ a2 ∧ a1
↠ (Handle-Performance ×2)
[unvalued(h)] ; [] ‖ true ∧ true
↠ (Simplify)
[unvalued(h)] ; [] ‖ true
↠ (Handle-Final)
[] ; [] ‖ true
```

1.3.3.5 Handling alternating possibilities

A more sophisticated handler.

Listing 1.19: Handler for alternating between heads and tails

```
term alternating-between-heads-and-tails (b-init : boolean)
  : Coin → boolean ↘ boolean
  := handler
      { #flip _ k ⇒ (b ⇒ k b (not b))
      ; value x   ⇒ (b ⇒ x)
      ; final f   ⇒ f b-init }.
```

Listing 1.20: Handle experiment with alternating between heads and tails.

```
term h := alternating-between-heads-and-tails true coin.

[] ; [] ‖ experiment with h
↠ (Handle)
[h] ; [] ‖ (coin#flip •) ∧ (coin#flip •)
↠ (Perform ×2)
```

```
[h] ; [(coin#flip •, a2),(coin#flip •, a1)] ‖ a2 ∧ a1
↠ (Handle-Value)
[unvalued(h)] ; [(coin#flip •, a2) ▹ (coin#flip •, a1)] ‖ (b1 ⇒ a2 ∧ a1)
↠ (Handle-Performance)
[unvalued(h)] ; [(coin#flip •, a1)]
  ‖ (_ k ⇒ (b2 ⇒ k b2 (not b2))) • (a2 b1 ⇒ a2 ∧ a1)
↠ (Simplify)
[unvalued(h)] ; [(coin#flip •, a1)] ‖ b2 ⇒ b2 ∧ a1
↠ (Handle-Performance)
[unvalued(h)] ; []
  ‖ (_ k ⇒ b3 ⇒ k b3 (not b3)) • (a1 b2 ⇒ b2 ∧ a1)
↠ (Simplify)
[unvalued(h)] ; [] ‖ b3 ⇒ (not b3) ∧ b3
↠ (Handle-Final)
[] ; [] ‖ (f ⇒ f true) (b3 ⇒ (not b3) ∧ b3)
↠ (Simplify)
[] ; [] ‖ false ∧ true
↠ (Simplify)
[] ; [] ‖ false
```

1.3.4 I/O

1.3.4.1 Resource for I/O

Listing 1.21: Resource for I/O

```
resource IO
  { output : string ↗ unit
  ; input  : unit ↗ string }.
```

1.3.4.2 Channel for I/O

Has language-implementation-specific link to operating system's standard I/O.

Listing 1.22: Channel for I/O

```
channel standard-io : IO.
```

1.3.4.3 Experiment for I/O

Listing 1.23: Experiment for I/O

```
term greetings : unit
  := do
      { name ← standard-io#input •
      ; standard-io#output ("Hello, " ++ name) }.
```

1.3.4.4 Handler for I/O

Must be defined primitively to allow for language-implementation-specific link to foreign calls to system's standard I/O.

Listing 1.24: Handler for standard I/O

```
primitive term standard-io-handler (α:Type) : α ↗ α.
```

We shall reference this appeal to a foreign I/O interface in the same way that 𝔹 and ℂ did. The interface $\mathcal{IO}$ responds to queries with the appropriate responses, matching the specification of the `io` resource. In this way the body of `standard-io-handler` can be imagined as the following:

Listing 1.25: Imaginary body for `standard-io-handler`

```
handler
  { #output s k ⇒ k 𝓘𝓞(output s)
  ; #input  _ k ⇒ k 𝓘𝓞(input •)
  ; value     a ⇒ a
  ; final     a ⇒ a }
```

Listing 1.26: Handle greetings with standard I/O

```
h := standard-io-handler

[] ; [] ‖ greetings with h
(Simplify)
[] ; []
  ‖ do { name ← standard-io#input •
       ; standard-io#output ("Hello, " ⧺ name) } with h
(Handle)
[h] ; []
  ‖ do { name ← standard-io#input •
       ; standard-io#output ("Hello, " ⧺ name) }
(Perform)
[h] ; [(standard-io#input •, a1)]
‖ do { name ← a1
     ; standard-io#output ("Hello, " ⧺ name) }
(Simplify)
[h] ; [(standard-io#input •, a1)] ‖ standard-io#output ("Hello, " ⧺ a1)
(Perform)
[h] ; [(standard-io#output ("Hello, " ⧺ a1),, a2)
    ▷ (standard-io#input •, a1)]
  ‖ a2
(Handle-Value)
[unvalued(h)] ; [(standard-io#output ("Hello, " ⧺ a1),, a2)
              ▷ (standard-io#input •, a1)]
  ‖ a2
(Handle-Performance)
[unvalued(h)] ; [(standard-io#output ("Hello, " ⧺ a1),, a2)]
  ‖ (x k ⇒ k 𝒹𝒪_1(input x)) • (a1 ⇒ a2)
(Simplify)
[unvalued(h)] ; [(standard-io#output ("Hello, " ⧺ a1),, a2)]
  ‖ (a1 ⇒ a2) 𝒹𝒪_1(input •)
(Handle-Performance)
[unvalued(h)] ; []
  ‖ (x k ⇒ k 𝒹𝒪_2(output x)) ("Hello, " ⧺ a1) (a2 ⇒ (a1 ⇒ a2) 𝒹𝒪_1(input •))
(Simplify)
[unvalued(h)] ; [] ‖ (a2 ⇒ (a1 ⇒ a2) 𝒹𝒪_1(input •)) 𝒹𝒪_2("Hello, " ⧺ a1)
(Simplify)
[unvalued(h)] ; [] ‖ (a1 ⇒ 𝒹𝒪_2("Hello, " ⧺ a1)) 𝒹𝒪_1(input •)
(Simplify)
[unvalued(h)] ; [] ‖ 𝒹𝒪_2("Hello, " ⧺ 𝒹𝒪_1(input •))
(Handle-Final)
```

```
[] ; [] ‖ 𝒥𝒪₂("Hello, " ++ 𝒥𝒪₁(input •))
```

Note that the usage of $\mathcal{IO}$ is subscripted by the order of use.

1.4 Considerations for Algebraic Effect Handlers

[**TODO**] Advantages:

1. unwrapped effect performances
2. easily nested effects
3. convenient separation of effects and handlers
4. TODO: look in algebraic effect handling papers for details

[**TODO**] Disadvantages:

1. loses type-checking effect handling
2. unhandled effects cause errors
3. certain kinds of co-recursive effects can cause issues
4. TODO: look in algebraic effect handling papers for details