K 1

Algebraic Effect Handlers

1.1 Introduction to Algebraic Effect Handlers

In chapter **??**, we considered 𝔹, an extension of 𝔸, that implemented effects by introducing specific language features for each kind of effect. While this allows simple reasoning about the behavior of those effects in 𝔹, it establishes no common reasoning about effects in general. If 𝔹 were extended to implement another effect using a new language feature, none of what is defined in 𝔹 explains how it should be implemented how it might behave.

What we desire is an extension to 𝔸 that provides a language feature for generally defining effects. In order to allow for the full scope of effects we are interested in, such an extension should implement two features:

- Effects may be defined as "black boxes" with implicit behavior defined outside the language. For example, the *IO* effect must appeal to an *IP* interface at some point, just like in 𝔹.
- Effects may be defined completely within the language. For example, the *state* and *exception* effects can be compeltely modeled within the language (e.g. chatper **??**).

In addition, we endeavor to achieve the following improvement over 𝔸's monadic effects:

- Effects are type-relevant.
- Effects are composable.

Algebraic effect handlers are such an extension to 𝔸 that meets all of these expectations. Its approach breaks the structure of effects into two parts:

- **Performances:** The code that corresponds to the *performing* of an effect. Performances are sensative to the whole program context.
- **Handlers:** The code that corresponds to the result of an effect performance, parametrized by the *handler*'s clauses. The handler is not sensative to the whole program context, and in its definition abstracts the context relevant to handling the performances (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.

- **Resources:** The code that corresponds to an instantiation of a specification of effects that are available to be performed and handled. The primitive effects it provides are called *actions*.

1.2 Language ℂ

Language ℂ implements algebraic effect handlers similarly to the scheme presented in [1].

1.2.1 Syntax for ℂ

Π 1.1: Syntax for ℂ

metavariable	**constructor**	**name**
«*Type*»	`Resource`	resource
«*type*»	`resource{ [` «*action-name*» `:` «*type*» ↗ «*type*» `;] }`	resource

	`handler`
	`{ [` «*term*»`#`«*action-name*» «*term-param*» «*term-param*» ⇒ «*term*» `;]`
	`; value` «*term-param*» «*term-param*» `=>` «*term*»
	`; finally` «*term-param*» ⇒ «*term*» `}`
		handler

Note that the `value` and `finally` clauses of the `handler` term-construct are optional. Their default implementations are given by the following notation:

1.2.2 Primitives for ℂ

Resource

[**TODO**] description

Action

[**TODO**] description

Listing 1.1: Notation for minimal handlers.

```
handler{ [ «term»#«action-name» «term-param» «term-param» ⇒ «term» ;] }

::=

handler
  { [ «term»#«action-name» «term-param» «term-param» ⇒ «term» ;]
  ; value a k ⇒ k a
  ; finally b ⇒ b }
```

Listing 1.2: Primitives for resources.

```
primitive term new ρ( : Resource) : ρ.
```

Performance

[**TODO**] description

Handling

[**TODO**] description

Sequencing

[**TODO**] description

Listing 1.3: Primitives for actions.

```
primitive type action : Type → Type → Type.
```

Listing 1.4: Notation for action.

```
«type»₁ ↗ «type»₂    ::=    action «type»₁ «type»₂
```

Listing 1.5: Primitives for performance.

```
primitive term perform ρ( : Resource)
  : ρ → action α β → α → performance β.
```

Listing 1.6: Notation for performance.

```
«term»₁#«term»₂    ::=    perform «term»₁ «term»₂
```

Listing 1.7: Primitives for handling.

```
primitive type handling : Type → Type → Type.

primitive term handle α( β : Type) : handling α β → α → β.
```

Listing 1.8: Notation for handling.

```
«type»₁ ↘ «type»₂    ::=    handling «type»₁ «type»₂
```

Listing 1.9: Notations for handling.

```
with «term»₁ do «term»₂    ::=    handle «term»₁ «term»₂

do «term»₁ with «term»₂    ::=    handle «term»₂ «term»₁
```

Listing 1.10: Primitives for sequencing

```
primitive term sequence : α → β → β.
```

Listing 1.11: Notation for sequencing.

```
«term»₁ >> «term»₂    ::=    sequence «term»₁ «term»₂
```

1.2.3 Typing Rules in ℂ

Π 1.2: Typing in ℂ

$$\textsc{Resource}\quad \frac{\begin{array}{l} \rho \coloneqq \texttt{resource\{ } [\ e_i : (\alpha_i \nearrow \beta_i)\ \ ;]\ \texttt{\}} \\ \Gamma \vdash \alpha_i : \texttt{Type} \quad (\forall i) \\ \Gamma \vdash \beta_i : \texttt{Type} \quad (\forall i) \end{array}}{\Gamma \vdash (\texttt{resource\{ } [\ e_i\ :\ \alpha_i \nearrow \beta_i\ \ ;]\ \texttt{\}}) : \texttt{Resource}}$$

$$\textsc{Perform}\quad \frac{\begin{array}{l} \rho \coloneqq \texttt{resource\{}\ldots\ e_i : \alpha_i \nearrow \beta_i\ \ldots\texttt{\}} \\ \Gamma \vdash r : \rho \\ \Gamma \vdash e_i : (\alpha_i \nearrow \beta_i) \\ \Gamma \vdash a : \alpha_i \end{array}}{\Gamma \vdash (r\#e_i\ \ a) : \beta}$$

$$\textsc{Handler}\quad \frac{\begin{array}{l} \rho \coloneqq \texttt{resource\{ } [\ e_i : (\alpha_i \nearrow \beta_i)\ \ ;]\ \texttt{\}} \\ \Gamma \vdash \rho : \texttt{Resource} \\ \Gamma \vdash r : \rho \\ \Gamma, a_i : \alpha_i, k_i : (\beta_i \to \beta) \vdash b : \beta \quad (\forall i) \\ \Gamma, a_v : \alpha \vdash b_v : \beta \\ \Gamma, b_f : \beta \vdash c_f : \gamma \end{array}}{\begin{array}{l} \Gamma \vdash (\texttt{handler\{ } [\ r\#e_i\ \ a_i\ \ k_i \Rightarrow b_i\ \ ;] \\ \qquad\qquad ;\ \texttt{value}\ a_v \Rightarrow b_v \\ \qquad\qquad ;\ \texttt{finally}\ b_f \Rightarrow c_f\ \ \texttt{\}}) : \alpha \searrow \gamma \end{array}}$$

1.2.4 Reduction Rules for $\mathbb{C}$

> [TODO] Since Handle-Effect only works on statements that aren't the *last* effect, there's a notation that appends a trivial ending to anything of that form.
>
> [TODO] need to rephrase these in terms of pushing the handlers onto a stack since can have nested handlers

The evaluation context notation $h, \mathcal{H}$ indicates that h is the top-most handler in the hander stack that handles the effect at hand. This breaks into two cases:

- For performances of actions from resource r, h is the top-most handler for r.
- For values, h is just the top-most handler.

Π 1.3: Reduction in $\mathbb{C}$

Do
$$\mathcal{H} \parallel \texttt{do}\ a\ \texttt{with}\ h \twoheadrightarrow h, \mathcal{H} \parallel a$$

Sequence
$$\frac{\text{value } v}{\mathcal{H} \parallel v \texttt{ >> } k \twoheadrightarrow \mathcal{H} \parallel k}$$

Handle-Effect
$$\frac{h \coloneqq \texttt{handler\{}\ldots\ r\#e_i\ a_i\ k_i \Rightarrow b_i\ \ldots\texttt{\}} \qquad \text{value } v}{\begin{array}{l} h, \mathcal{H} \parallel \texttt{let}\ x \coloneqq r\#e_i\ v\ \texttt{in}\ k \twoheadrightarrow \\ h, \mathcal{H} \parallel (a_i\ k_i \Rightarrow b_i)\ v\ (x \Rightarrow k) \end{array}}$$

Handle-Value
$$\frac{h \coloneqq \texttt{handler\{}\ldots\ \texttt{value}\ a_v \Rightarrow b_v\ \ldots\texttt{\}} \qquad \text{value } v}{\begin{array}{l} h, \mathcal{H} \parallel \texttt{let}\ x \coloneqq v\ \texttt{in}\ k \twoheadrightarrow \\ h, \mathcal{H} \parallel (a_v\ k_v \Rightarrow k_v\ b_v)\ v\ (x \Rightarrow k) \end{array}}$$

Handle-Finally
$$\frac{h \coloneqq \texttt{handler\{}\ldots\ \texttt{finally}\ b_f \Rightarrow c_f\ \ldots\texttt{\}} \qquad \text{value } v}{h, \mathcal{H} \parallel v \twoheadrightarrow \mathcal{H} \parallel (b_f \Rightarrow c_f)\ v}$$

1.3 Examples

1.3.1 Example: Nondeterminism

```
// specify a resource for coin-flipping effect
// flip returns true if heads and false if tails
type coin-flipping : Resource
  ≔ new resource{ flip : unit → boolean }.

// create a new resource instance of the coin-flipping effect
term coin : coin-flipping ≔ new coin-flipper.

count (b : boolean) ≔ if b then 1 else 0.

// a term that uses coin to perform the coin-flipping effect
term experiment : coin#:(unit ↗ integer) ≔
  let x1 ≔ coin#flip • in
  let x2 ≔ coin#flip • in
  count x1 + count x2.
```

```
// a handler that accumulates all possible results of experiment
term accumulate : coin#:(integer ↗ list integer) ≔
  handler{ coin#flip - k ⇒ k true <> k false
         ; value x ⇒ [x] }.

// accumulate results of experiment
do experiment with accumulate
```

```
// reduction:
do experiment with accumulate
 ↠
with accumulate do
  let x1 ≔ coin#flip • in
  let x2 ≔ coin#flip • in
  count x1 + count x2
 ↠
with accumulate do
  ( (_ k ⇒ k true <> k false) •
    (x1 ⇒ let x2 ≔ coin#flip • in
          count x1 + count x2) )
 ↠
with accumulate do
  ( let x2 ≔ coin#flip • in
    count true + count x2 )
  <>
  ( let x2 ≔ coin#flip • in
    count false + count x2 )
 ↠
with accumulate do
  ([count true + count true] <> [count true + count false]) <>
  ([count true + count true] <> [count true + count false])
 ↠
with accumulate do [[2, 1], [1, 0]]
 ↠
[[[2, 1], [1, 0]]]
```

B

[1] Bauer, A., & Pretnar, M. (2015). Programming with algebraic effects and handlers. *J. Log. Algebraic Methods Program.*, *84*, 108–123.