# Purity and Effect

---

A Thesis
Presented to
The Division of Mathematics and Natural Sciences
Reed College

---

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Arts

---

Henry Blanchette

Approved for the Division
(Computer Science)

---

Jim Fix

# Π

# K 0

# Conventions

## 0.1 Languages

The discussion of formal languages requires the use of several different languages simultaneously. The following is a list of the languages used in this work:

- **narrative (informal) language**: The top-level language used to narrate this work in an intuitive prose; English.
- **logical meta-language**: Formalized expressions that make no extra-logical assumptions. Contains expressions such as "The proposition $A$ implies the proposition $A$."
- **mathematical meta-language**: Mathematical expressions of generalized programs, where terms range more freely than in actual valid programs in the programming language. Relies on a mathematical context not explicit in this work. Contains expressions such as "$f : \alpha \to \beta \to \gamma$."
- **programming language**: Programs that are fully defined by the contents of this work, relying on no external context. Necessarily conforms to the given syntax, and its expressions have no meaning beyond the given rules that apply to them. Contains expressions such as "`term id (α:Type) (a:α) : α := a.`"

## 0.2 Fonts

The use of many different languages simultaneously, as described by the previous section, has the unfortunate consequence of certain expressions seeming ambiguous to the reader. In order to mitigate this problem, each language and certain kinds of phrases are designated a font. The following fonts are designated in this way:

- **normal font**: narrative language.
- **italic font**: emphasis in narrative language.

- **bold font**: introductory use of new terms in narrative language, and syntactic structures in narrative language.
- **small-caps font**: names in logical meta-language.
- **sans-serif font**: mathematical meta-language.
- **monospace font**: programming language.

## 0.3 Names

Naming conventions:

[TODO] Finalize

- **terms**: lower-case english word/phrase.
- **term variables**: lower-case english letter.
- **types**: lower-case english word/phrase.
- **types of higher order**: capitalized english word/phrase.
- **type variables**: lower-case greek letter.
- **type variables of higher order**: capital english letter.

# K 1

# Introduction

[**TODO**] change “Type” to “kind”

[**TODO**] change declaration header ‘term‘ and ‘type‘ to just one keyword, say ‘let‘

[**TODO**] is this entire chapter, change concrete programs to be entirely monospace font

[**TODO**] include note or something explaining what the convention is for reduction and other rules that have the same name (e.g. SIMPLIFY)

[**TODO**] introduce terms **compile-time** and **run-time** somwhere. Or, come up with new terms, like **typing** and **evaluation** perhaps? But doesn't exactly capture everthing. Probably better: **compile-time** and **evaluation-time**.

[**TODO**] fix style for colon, I think its too far apart (especially around commas): $a : \alpha$

## 1.1 Introduction

[**TODO**] Introduce my thesis.

## 1.2 Language $\mathbb{A}$

lambda-calculus is a formal language for expressing computation. In this context, a **language** is defined to be a set of expression that are generated by syntactical rules. The lambda-calculus comes in many different variants, and here we will consider a basic variant of the **simply-typed lambda-calculus** in order to considering computation formally. Call our language $\mathbb{A}$.

### 1.2.1 Syntax for 𝔸

In 𝔸, there are two forms used for constructing well-formed expressions: **terms** and **types**. They are expressed by the following syntax:

Π 1.1: Syntax for 𝔸

| **metavariable** | **constructor** | **name** |
|---|---|---|
| «*program*» | ⟦ «*declaration*» ⟧ | program |
| «*declaration*» | `type` «*type-name*» `:` «*kind*» `:=` «*kind*»`.` | constructed type |
| | `primitive type` «*type-name*» `:` «*kind*»`.` | primitive type |
| | `term` «*term-name*» `:` «*type*» `:=` «*term*»`.` | constructed term |
| | `basic term` «*term-name*» `:` «*type*» `:=` «*term*»`.` | basic term |
| | `primitive term` «*term-name*» `:` «*type*»`.` | primitive term |
| «*kind*» | `type1` | atom |
| | `type1 →` «*kind*» | arrow |
| «*kind*» | «*type-name*» | atom |
| | `(`«*type-param*» `:` «*kind*»`) ⇒` «*kind*» | function |
| | «*kind*» «*kind*» | application |
| «*term*» | «*term-name*» | atom |
| | `(`«*term-param*» `:` «*kind*»`) ⇒` «*term*» | function |
| | «*term*» «*term*» | application |

#### 1.2.1.1 Metavariables and Names

The metavariables «*term-name*», «*type-name*», and «*term-param*» range over, for a given program, a fixed and collection of names. There are two kinds of names included in this collection:

- **constructed names**: Require a construction written in the language; establish a *functional equality* between name and construction — the name universally can be substituted *with* its construction.

- **basic names**: Require a construction written in the language; establish a *definitional equality* between name and construction — the name universally can be substituted *with* or *for* its construction.

- **primitive names**: Included a priori. Terms and types provided this way are provided along with their a priori type or kind respectively, and primitive terms along with a priori reduction rules (if any). Primitive names are also trivially basic.

This syntax schema is formatted as a *generative context-free grammar*, with

- *non-terminals*: The meta-variables «*progam*», «*declaration*», «*kind*», «*kind*», «*term*».

- *terminals*: Ranged over by the meta-variables «*term-name*», «*type-name*».

- *repeated*: Expressions of the form `[` «*meta-var*» `]` indicate that any number of «*meta-var*» can be repeated in its place. Note also that metavariables inside of repeated expressions may be indexed by $i$ or $j$, indicating that they are the $i$-th repeated sub-expression.

The following are some examples of the sorts of formal items that these meta-variables stand range over:

- «*term-name*»: `1`, `12309`, `true`, `false`, `"hello world"`, `•`.

- «*type-name*»: `natural`, `integer`, `boolean`, `string`, `void`, `unit`.

- «*term-param*»: `x`, `y`, `this-is-a-parameter`.

[**TODO**] include some function terms and (and maybe types?)

A given expression is well-formed with respect to 𝔸 if it is constructible by a series of applications of these rules. Note that this syntax does not specify any concrete terms or types. In this way, 𝔸's syntax is defined so abstractly in order to make the definitions for its typing and semantics as concise as possible. For example, we shall reference to the type `int`, some «*term-name*»s that have type `int` such as `0`, `1`, `-1`, `2`, `-2`, `3`, `-3`, and so on. Such types and terms as these are called **atoms** because they do not have any internal; they are simply posited as primitives in the language.

### 1.2.2 Notations for 𝔸

Notations establish a convenient shorthand for certain structures that establish a **syntactical equivalency** — the notation and the structure is abbreviates are universally substitutable for each other. In the programming languages community such these notations are commonly referred to as **syntax sugar**, since they are addictive and can concisely and readable express otherwise very dense code. Notations posit *new* structures in the syntax of 𝔸, but they can completely reduce to the core set of syntactical structures as defined by table **??**. Since there are some code structures that will be used very commonly throughout this work, we shall adopt a variety of notations. This subsection starts us off with with the most common and low-level notations.

> [**TODO**] describe notational convention for lists of joined items i.e. having "[[ i ; ]]" means that the ";" goes between each of the "i"s in the list.

#### 1.2.2.1 Declarations

> [**TODO**] define how declarations work, and are not part of evaluation exactly...

> [**TODO**] give typing rules for declarations

#### 1.2.2.2 Parameters

For parametrized terms and types, a convenient notation is to accumulate the parameters to the left side of a single " ⇒ " as follows:

Listing 1.1: Notation for multiple parameters

```
(«term-param»_1 : «kind»_1) ⋯ («term-param»_n : «kind»_n) ⇒ «term»_*
  ::=
    («term-param»_1 : «kind»_1) ⇒ ⋯ ⇒ («term-param»_n : «kind»_n) ⇒ «term»_*

(«type-param»_1 : «kind»_1) ⋯ («type-param»_n : «kind»_n) ⇒ «kind»_*
  ::=
    («type-param»_1 : «kind»_1) ⇒ ⋯ ⇒ («type-param»_n : «kind»_n) ⇒ «kind»_*
```

When defining a term or type in a declaration it is convenient to write the names of parameters immediately to the right of the name being defined, resembling the syntax for applying the new term or type to its given arguments. The following notations implement this.

Listing 1.2: Notation for declaration parameters

```
term «term-name» ⟦ («type-param»:«kind») ⟧ ⟦ («term-param»:«type») ⟧
```

```
  : «type» := «term».
    ::=
      term «term-name»
        : ⟦ («type-param»:«kind») ⟧ ⇒ «type»
        := ⟦ («term-param»:«type») ⟧ ⇒ «term».

type «type-name» ⟦ («type-param»:«kind») ⟧ : «kind» := «type».
  ::=
    type «type-name» : «kind» := ⟦ («type-param»:«kind») ⟧ ⇒ «type».
```

When two consecutive parameters have the same type or kind, the following notation allows a reduction in redundancy:

Listing 1.3: Notations for multiple shared-type and shared-kind parameters

```
(⟦ «term-param» ⟧ : «type») ⇒    ::=    ⟦ («term-param»:«type») ⟧ ⇒

(⟦ «type-param» ⟧ : «kind») ⇒    ::=    ⟦ («type-param»:«kind») ⟧ ⇒
```

#### 1.2.2.3 Local bindings

This is a core feature in all programming languages, and is expressed in 𝔸 with this notation:

Listing 1.4: Notation for local binding.

```
let «term-param»₁ : «kind»₁ := «term»₂ in «term»₃
  ::=
    ((«term-name»₁:«term»₁) ⇒ «term»₃) «term»₂
```

Such a binding allows for the scoped binding of a name to a value. The instance of the name introduced is only available from inside «*term*»$_3$ — the *body* of the local binding.

#### 1.2.2.4 Omitted types

The types of «*term-param*»s may sometimes be omitted when they are unambiguous and obvious from their context. For example, in

```
term twice : (α → α) → α → α :=
  f a ⇒
    let a' = f a in
    f a'.
```

the types of `f`, `a`, and `a'` are obvious from the immediately-previous type of `twice`.

### 1.2.3 Primitives in 𝔸

The primitive names for a program are defined by its `primitive term` and `primitive type` declarations, and the constructed names for a program are defined by the other `term` and `type` declarations. There are three particularly prevalent types, along with some primitive terms, to have defined as part of the core of 𝔸. They are the arrow type, sum type, and product type.

#### 1.2.3.1 Arrow Type

The first notable primitive type is the *arrow type*, where `arrow` $\alpha$ $\beta$ is the type of terms that are functions with input type $\alpha$ and output type $\beta$. All we need is the following declaration, since functions are a syntactic structure already introduced by table **??**.

Listing 1.5: Primitive for the arrow type

```
primitive type arrow : type1 → type1 → type1.
```

##### 1.2.3.1.1 Notation for infixed type arrow.

[**TODO**] english

Listing 1.6: Notation for infixed type arrow

«*kind*»$_1$ → «*kind*»$_2$ ::= `arrow` «*kind*»$_1$ «*kind*»$_2$

This notation is right-associative. For example, the types $\alpha \to \beta \to \gamma$ and $\alpha \to (\beta \to \gamma)$ are equal. This right-associativity corresponds to the idea of *currying* functions (also called *partial application*). For example consider what the type of a function, that takes two inputs of types $\alpha$ and $\beta$ respectively and has output of type $\gamma$, should be. The type could be written `product` $\alpha$ $\beta$ → $\gamma$ (the `product` type is defined in the soon subsubsection "Product Type"), where `product` $\alpha$ $\beta$ contains the two inputs. It could also be written $\alpha \to \beta \to \gamma$, which associates to $\alpha \to (\beta \to \gamma)$,

#### 1.2.3.2 Sum Type

The second notable primitive type is the *sum type*, where `sum` $\alpha$ $\beta$ is the type of terms that are either of type $\alpha$ (exclusively) or of type $\beta$. This property is structured by the following declarations:

Listing 1.7: Primitives for the sum type

```
primitive type sum : type1 → type1 → type1.

// constructors
primitive term left  (α β : type1) : α → sum α b.
primitive term right (α β : type1) : β → sum α b.

// destructor
primitive term split (α β γ : type1)
  : sum α β → (α → γ) → (β → γ) → γ.
```

#### 1.2.3.2.1 Notation for infixed type sum.

[TODO] english

Listing 1.8: Notation for infixed type sum

«*kind*»$_1$ ⊕ «*kind*»$_2$ ::= sum «*kind*»$_1$ «*kind*»$_2$

This notation is right-associative. For example, the types $\alpha \oplus \beta \oplus \gamma$ and $\alpha \oplus (\beta \oplus \gamma)$ are equal.

#### 1.2.3.2.2 Notation for cases of a sum term.

[TODO] more english

We also introduce the following syntax sugar for `split` — the following `cases` notation:

Listing 1.9: Notation for `case`.

```
cases «term»* to «kind»*
  { left («term-param»1:«kind»1) ⇒ «term»1
  ; right («term-param»2:«kind»2) ⇒ «term»2 }

::=

split «kind»1 «kind»2 «kind»*
  «term»*
  ((«term-param»1:«kind»1) ⇒ «term»1)
  ((«term-param»2:«kind»2) ⇒ «term»2)
```

Note however that the `to` «*kind*»$_*$ phrase in this notation will often omitted and leave the type of the case expression implicit (in the same way as described by paragraph 1.2.2.4).

**1.2.3.2.3 *n*-ary sums.** The definition of `sum` can be considered as a special case of *n*-ary sum types — the binary sum type. However only the binary case need be introduced primitively, since any *n*-ary sum type can be constructed by a nesting of binary sum types. For example, the sum of types $\alpha, \beta, \gamma$ can be written `sum α (sum β γ)`. For example, the product of types $\alpha, \beta, \gamma$ can be written `α ⊕ (β ⊕ γ)`. To streamline representation, we adopt that $\times$ is right-associative; the product `α ⊕ (β ⊕ γ)` can be written simply as `α ⊕ β ⊕ γ`. Observe that providing a $b : \beta$ as a case of this sum is cumbersomely written `right (left b)`. The following notation specifies a family of terms of the form `case-`*i* where $0 < i \in \mathbb{Z}$, which construct the *i*-th component of a sum term.

```
case-i   ::=   (right ∘ ⋯ (i−1) ∘ right ∘ left)
```

Finally, destructing an *n*-ary product involves many levels of `cases` where each level handles just two cases at a time. The following notation flattens this structure by allowing the top-level `cases` to have more than two branches.

Listing 1.10: Notation for destructing *n*-ary sum types

```
cases «term»* to «kind»*
  { case-1 («term-param»1:«kind»1) ⇒ «term»1
  ; ⋯
  ; case-n («term-param»n:«kind»n) ⇒ «term»n }

::=

split «kind»1 «kind»2 «kind»*
  «term»*
  («term-param»1:«kind»1) ⇒ «term»1)
  (x ⇒
    cases x to «kind»*
      { case-1 («term-param»2:«kind»2) ⇒ «term»2
      ; ⋯
      ; case-(n − 1) («term-param»n:«kind»n) ⇒ «term»n })
```

This notation is defined inductively for the sake of simplicity. Note that the parameter *x* introduced is a **fresh** name i.e. unique and not able to be referenced from outside this notation definition.

**1.2.3.2.4 Named sums.** The sum type is widely applicable for modeling data types which have terms that must be exclusively one from a delimited set of cases. Such data types can be defined as the sum of the types of each case. In such constructions, it is convenient to name the components as basic terms. The following notation provides a conceptually-fluid way of defining them.

Listing 1.11: Notation for defining named sum types and constructing named sum terms

```
type «type-name»* : «kind»*
  := «term-name»1:«type»1 | ⋯ | «term-name»n:«type»n.

::=

type «type-name»* : «kind»* := «type»1 ⊕ ⋯ ⊕ «type»n.

// constructors
basic term «term-name»1 : «type»1 → «type-name»* := case-1.
⋮
basic term «term-name»n : «type»n → «type-name»* := case-n.
```

There is an additional requirement for this notation that kinds $\text{«}kind\text{»}_1, \ldots, \text{«}kind\text{»}_n$ are each (independently) one of the following: either $\text{«}type\text{»}_*$ or an application of $\text{«}type\text{»}_*$, or, an $n$-ary arrow type ending with either $\text{«}type\text{»}_*$ or an application of $\text{«}type\text{»}_*$.

As for destruction, the notation given in listing 1.10 is compatible with replacing the `case -i` with the constructors named by the named sum type definition since the constructors are basic terms.

### 1.2.3.3 Product Type

The third notable primitive type to define here is the product type, where `product α β` is the type of terms that are a term of type $\alpha$ joined with a term of type $\beta$. This property is structured by the following declarations:

Listing 1.12: Primitives for `product`.

```
primitive type product : type1 → type1 → type1.

// constructor
primitive term pair (α β : type1) : α → β → product α β.

// destructors
primitive term first (α β : type1) : product α β → α.
```

```
primitive term second (α β : type1) : product α β → β.
```

#### 1.2.3.3.1 Infixed type product.

[TODO] english

Listing 1.13: Notation for infixed type product

```
«kind»₁ × «kind»₂   ::=   product «kind»₁ «kind»₂
```

This notation is right-associative. For example, the types $\alpha \times \beta \times \gamma$ and $\alpha \times (\beta \times \gamma)$ are equal.

**1.2.3.3.2 $n$-ary products.** The definition of `product` can be considered as a special case of $n$-ary product types — the binary product type. However only the binary case need be introduced primitively, since any $n$-ary product type can be constructed by a nesting of binary product types (this is the same strategy as for $n$-ary sum types before). For example, the product of types $\alpha, \beta, \gamma$ can be written `α × (β × γ)`. However, constructing a product term from $a : \alpha, b : \beta, c : \gamma$ is cumbersomely written as `pair a (pair b c)`. A common mathematical and computer science notation for product terms, often called *tuples*, is the following.

Listing 1.14: Notation for constructing $n$-ary product terms

```
(«term»₁, ..., «term»ₙ)
  ::=
    pair «term»₁ (pair ⋯ (pair «term»ₙ₋₁ «term»ₙ))
```

Additionally, the projecting of an $n$-ary product term onto one of its components is verbose. For example, the function that projects a term of type $\alpha \times \beta \times \gamma \times \delta$ onto its second component is constructed `second ∘ second ∘ first`. The following notation specifies a family of terms of the form `part-`$i$ where $0 < i \in \mathbb{Z}$, which destruct a product term by projecting onto its $i$-th component.

Listing 1.15: Notation for destructing $n$-ary product terms

```
part-i   ::=   (second ∘ ⋯^(i−1) ∘ second ∘ first)
```

**1.2.3.3.3 Named products.** The product type is widely applicable for modeling data types which are composed of several required parts. Such data types can be defined as the product of the types of each other their parts. In such constructions, it is convenient to name the parts (in a similar way to the cases of named sums). These constructions are often called *record types*, since they liken to a record holding named data entries. The following notation provides a conceptually-fluid way of defining them.

Listing 1.16: Notation for defining named products types and constructing named products terms

```
type «type-name»_* : «kind»_*
  := { «term-name»_1 : «type»_1 ; ⋯ ; «term-name»_n }.

::=

type «type-name»_* : «kind»_* := «type»_1 × ⋯ × «type»_n.

// destructors
term «term-name»_1 : «type-name»_* → «type»_1 := part-1.
⋮
term «term-name»_n : «type-name»_* → «type»_n := part-n.
```

**1.2.3.3.4 Other Common Primitives.**

[TODO] mention other common primitives that will not be explicitly defined, based on the following:

- integer
- natural
- boolean
- unit

[TODO] note that primitive terms need to have reduction rules defined outside of 𝔸 (if they have any reductions), and that these are not guaranteed to not break the rest of 𝔸. So separate proofs need to be provided for complicated primitive stuff.

**1.2.3.3.5 Infixed Associatvive Levels** Having multiple infixed notations for «*kind*»s introduce inter-notational ambiguity. For example, the type $\alpha \to \beta \times \gamma \oplus \delta$ has not yet been determined to associate in any one of many possible ways. We adopt the following precedence order of increasing tightness to eliminate this ambiguity: $\to$ , $\oplus$ , $\times$ . So, the type $\alpha \to \beta \times \gamma \oplus \delta$ associates as $(\alpha \to \beta) \times (\gamma \oplus \delta)$

### 1.2.4 Type application

The notation for «*type-name*» «*kind*» ⋯ «*kind*» is read as «*type-name*» acting as a kind of function that takes some number of type parameters. So, `arrow` is a type-constructor with type type parameters, say $\alpha$ and $\beta$, and forms the type `arrow α β`. Note that this is a different kind of function than the **function** syntactical construct, but the intuitive similarity will be addressed later in chapter **??**.

### 1.2.5 Typing Rules for 𝔸

[**TODO**] fix code in math mode

[**TODO**] write Typing rules for types

Terms and types are related by a **typing judgement**, by which a term is stated to have a type. The judgement that $a$ has type $\alpha$ is written as $a : \alpha$. In order to build typed terms using the constructors presented in **??**, the types of complex terms are inferred from their sub-terms using inference rules making use of judgement contexts (i.e. collections of judgements). A statement of the form $\Gamma \vdash a : \alpha$ asserts that the context $\Gamma$ entails that $a : \alpha$. The notation $\Gamma, J \vdash J'$ abbreviates $\Gamma \cup \{J\} \vdash J'$. Keep in mind that these propositions (e.g. judgements) and inferences are in an explicitly-defined language that has no implicit rules. The terms "inference" and "judgement" are called such in order to have an intuitive sense, but rules about judgement and inference in general (outside of this context) are not implied to also apply here.

With judgements, we now can state **typing inferences rules**. Such inference rules have the form which asserts that the premises $P_1, \ldots, P_n$ entail the conclusion $Q$. For example, could be a particularly uninteresting inference rule. Explicitly stating the domain of each variable as premises is cumbersome however, so the following are conventions for variable domains based on their name:

- $\Gamma, \Gamma_i, \ldots$ each range over judgement contexts,
- $a, b, c, \ldots$ each range over terms,
- $\alpha, \beta, \gamma, \ldots$ each range over types.

The following typing rules are given for 𝔸:

[**TODO**] describe kinding rules for kinds (of types)

Π 1.2: Typing in 𝔻: Resource

$$\textsc{TODO}\ \frac{\Gamma, a{:}\alpha \vdash a{:}\alpha}{a{:}\alpha}$$

$$\textsc{Function-Abstraction}\ \frac{\Gamma, a{:}\alpha \vdash b{:}\beta}{\Gamma \vdash ((a{:}\alpha) \Rightarrow b){:}(\alpha \to \beta)}$$

$$\textsc{Function-Application}\ \frac{\Gamma \vdash a{:}(\alpha \to \beta) \qquad \Gamma \vdash b{:}\alpha}{\Gamma \vdash (a\ b){:}\beta}$$

### 1.2.6 Reduction Rules for 𝔸

Finally, the last step is to introduce **reduction rules**. So far we have outlined syntax and inference rules for building expressions in 𝔸, but all these expressions are inert. Reduction rules describe how terms can be transformed, step by step, in a way that models computation. A series of these simple reductions may end in a term for which no reduction rule can apply. Call these terms **values**, and notate "$v$ is a value" as "value $v$." The following reduction rules are given for 𝔸:

Π 1.3: Reduction in 𝔸

$$\textsc{Simplify}\ \frac{a \twoheadrightarrow a'}{a\ b \twoheadrightarrow a'\ b}$$

$$\textsc{Simplify}\ \frac{b \twoheadrightarrow b' \qquad \text{value } v}{v\ b \twoheadrightarrow v\ b'}$$

$$\textsc{Apply}\ \frac{\text{value } v}{((a : \alpha) \Rightarrow b)\ v \twoheadrightarrow [v/a]\ b}$$

$$\textsc{Split-Right}\quad \texttt{split}\ \alpha\ \beta\ \gamma\ (\texttt{left}\ a)\ f\ g \twoheadrightarrow f\ a$$

$$\textsc{Split-Left}\quad \texttt{split}\ \alpha\ \beta\ \gamma\ (\texttt{right}\ a)\ f\ g \twoheadrightarrow f\ b$$

$$\textsc{Project-First}\quad \texttt{first}\ \alpha\ \beta\ (a, b) \twoheadrightarrow a$$

$$\textsc{Project-Second}\quad \texttt{second}\ \alpha\ \beta\ (a, b) \twoheadrightarrow b$$

[TODO] note that Apply is usually referred to as $\beta$-reduction.

The most fundamental of these rules is APPLY (also known as $\beta$-reduction), which is the way that function applications are resolved to the represented computation's output. The substitution notation $[v/a]b$ indicates to "replace with $v$ each appearance of $a$ in $b$." In this way, for a function $((a{:}\alpha) \Rightarrow b) : \alpha \to \beta$ and an input $v : \alpha$, $\beta$-Reduction **substitutes** the input $v$ for the appearances of the function parameter $a$ in the function body $b$.

For example, consider the following terms:

> [TODO] enlightening example of a series of $\beta$-reductions for some simple computation.

### 1.2.7 Properties of $\mathbb{A}$

With the syntax, typing rules, and reduction rules for $\mathbb{A}$, we now have a completed definition of the language. However, some of the design decisions may seem arbitrary even if intuitive. This particular framework is good because it maintains a few nice properties that make reasoning about $\mathbb{A}$ intuitive and extendable.

> [TODO] define these properties in English; want to keep prog-language and meta-language separate.

**Theorem 1.1. (Type-Preserving Substitution in $\mathbb{A}$).** If $\Gamma, a{:}\alpha \vdash b{:}\beta$, $\Gamma \vdash v{:}\alpha$, and value $v$, then $\Gamma \vdash ([v/a]b){:}\beta$.

**Theorem 1.2. (Reduction Progress in $\mathbb{A}$).** If $\{\} \vdash a{:}\alpha$, then either value $a$ or there exists a term $a'$ such that $a \twoheadrightarrow a'$.

**Theorem 1.3. (Type Preservation in $\mathbb{A}$).** If $\Gamma \vdash a{:}\alpha$ and $a \twoheadrightarrow a'$, then $\Gamma \vdash a'{:}\alpha$.

**Theorem 1.4. (Type Soundness in $\mathbb{A}$).** If $\Gamma \vdash a{:}\alpha$ and $a \twoheadrightarrow a'$, then either value $a'$ or there exists a term $a''$ such that $\Gamma \vdash a''{:}\alpha$ and $a' \twoheadrightarrow a''$.

**Theorem 1.5. (Strong Normalization in $\mathbb{A}$).** For any term $a$, either value $a$ or there is a sequence of reductions that ends in a term $a'$ such that value $a$.

# K 2

# A Simple Approach to Effects

[TODO] add captions to all listings

[TODO] change “Type” to “kind”

[TODO] use different symbol for collective of reduction contexts, since $\Delta$ is already used in other papers for linar logics

## 2.1 Declarative and Imperative Programming Languages

Previously in section 1.2, we defined $\mathbb{A}\backslash$ as a basic variant of the $\lambda$-calculus. In the general context of programming languages in general, $\mathbb{A}\backslash$ falls under the category of *declarative* as apposed to *imperative* programming languages. **Declarative** programming languages have a style focussed on *mathematically defining the result of computations* i.e. the logic rather that execution flow. **Imperative** programming languages, in contrast, have a style focussed on *instructing how computations should be carried out* i.e. the execution flow rather than logic. The differences between these kinds arise only in high-level programming languages, because low-level languages (assembly and C-level languages) are by definition instructing the computer what to do roughly step-by-step. Additionally, many languages implement features from both of these categories.

As an example consider the familiar task of summing a list of integers. A declarative algorithm to solve this task is the following.

> **The sum of a list of integers (Declarative).** Let $l$ be a given list of integers. If $l$ is empty, then the sum is $0$. Otherwise $l$ is not empty, so the sum is $h$ plus the sum of $t$, where $h$ is the head of $l$ and $t$ is the tail of $l$.

On the other hand, an imperative algorithm for the same task is the next.

> **The sum of a list of integers (Imperatively).** Let $l$ be a given list of

integers. Declare $s$ to be an integer variable[1] , and initialize $s$ to be $0$. Declare $i$ to be an integer variable, and initialize $i$ to be $1$. If $i$ is no greater than the length of $l$, then declare $x$ to be the $i$-th element of $l$ and set $s$ to $s + x$, then set $i$ to $i + 1$, then repeat this sentence; otherwise, the sum is $s$.

This example demonstrates the difference between the two styles. In the declarative style, the algorithm describes what the sum of a list of integers in terms of an inductive understanding of a list as either empty or a head integer and a tail list. Such an inductive structuring of a list naturally yields a recursive definition. On the other hand, in the imperative style, the algorithm describes how to keep track of the partial sum while stepping through the list. This naturally yields a definition with a conditional LOOP that repeats for each list element and a variable to store the partial sum.

[**TODO**] transition to list, and rewrite descriptions about effects since I haven't introduced that yet, and isn't even a core difference between them. Pad some English around it too.

**Declarative** programming language treat computation as the evaluation of stateless mathematical functions considered within a stateless mathematical context. Such languages often assume immutable names and explicit delineations between Examples: Scheme, Lisp, Standard ML, Clojure, Scala, Haskell, Agda, Gallina, Coq, Scala.

**Imperative** programming languages treat computation as a sequence of instructions carried out within a stateful execution context. Such languages typically assume mutable names and other stateful structures. Example: Smalltalk, Simula, Algol, Bash, Java, C, C++, Python.

However, these categories are not so strict and many languages borrow designs from both. For the most part, these categories are stylistic rather than strict. For example, many declarative languages have special support for mutable variables and other stateful structures. And, many high-level[2] imperative languages provide certain structures that mimic declarative definitions that may be translated to imperative definitions at a lower level.

[1] It is an unfortunate convention, outside this work, that all programmer-introduced names are referred to as *variables*. While it is true that the same name $x$ may be bound to different values in different algorithms, it is not necessarily true that $x$ may be mutated within the same algorithm. This is especially relevant to the declarative programming style where, in this terminology, typically *no variables are mutable* — an unfortunate and entirely avoidable clash of terms. A more consistent terminology defines a **name** to be reference to a value, an **immutable** name (or **constant**) to be a name who's value cannot be mutated, and a **mutable** name (or **variable**) to be a name who's value can be mutated. (Note that I am using "value" in a more generalized sense here than I use in the definition of reduction rules.) In order to set an example in this work, it shall use this terminology.

[2] High-level in terms of abstraction from directly interfacing with the computer, which always executes in an way paralleling an imperative description.

## 2.2 Computation with Effects

> [TODO] describe actual state of programming with effects i.e. writing C code (imperative).

It turns out that the well-known distinction between declarative and imperative programming languages maps onto a lesser-known distinction between effectual and pure programming languages. Notice that for terms in $\mathbb{A}$, reductions are **context independent** — the evaluation of a term relies only on information contained explicitly within the term.

> [TODO] convincing example of how $\mathbb{A}$ terms are context independent

In imperative languages, it is rarely the case that evaluation is context independent. As a simple example, just consider mutability. This Java program defines a class with two methods[3], `increment` and `triple`, that each perform a different mutation of the class's variable field[4], `x`.

```
class A {
  int x;

  // creates a new instance of A
  // with a field x set to the given x value
  public A(int x) {
    this.x = x;
  }

  // adds 1 to this A instance's field x
  public void increment() {
    this.x = this.x + 1;
  }

  // triples this A instance's field x
  public void triple() {
    this.x = this.x * 3;
  }
}
```

Given this setup, consider the following two functions that call `A`'s methods in different orders.

[3] In object-oriented programming lingo, the functions defined in an object class are called **methods**, which the class implements for each instance of the class.

[4] In object-oriented programming lingo, the non-function names defined in an object are called **fields**, which each class instance gets its own unique of.

```
public int f1() {
  A a = new A(0); // create a new A instance with its x field set to 0
  a.increment();  // increment firstly, then
  a.triple();     // triple secondly
  return a.x;     // return the value of a's x field
}

public int f1() {
  A a = new A(0); // create a new A instance with its x field set to 0
  a.triple();     // triple firstly, then
  a.increment();  // increment secondly
  return a.x;     // return the value of a's x field
}
```

Running `f1()` returns `3`, but running `f2()` returns `0`. This is because there is a background state being managed as the program is running — a state that is passed from execution step to execution step but not explicitly encapsulated by each step's programmatic expression. This state is an example of an *implicit context*. In general, an **implicit context** is an implicit set of information maintained during computation that is carried along from step to step, can be interacted indirectly by code using a special interface, and can affect the results of computation. The adjective "implicit" is important because it removes such a context from the usual computational contexts of set of names and value that can be directly referenced by code.

In more simple terms, implicit contexts are the execution-relevant contexts that are not directly accessible from within a program. In this way, we can say that the characteristic implicit contexts of declarative languages include execution order. Of course, all programming languages have many implicit contexts, many of them shared by most languages. For example, assembly languages explicitly keep track of registers and memory pointers, so they are among the explicit contexts. But most higher-level languages, though they ultimately compile to assembly code, do not allow reference to specific registers or memory pointers, so in higher-level languages they are among the implicit contexts.

In the early days of computer engineering, often a program was written to be run by a single, unique physical machine. Still today, a program is written as a code to be run on a physical machine[5]. The unique aspects of that machine still have an impact on how that running actually happens of course, but the development has been to increase the level of abstraction of the program, for example so that a program can be written in as a code to be run on many very different computers. This first level of abstraction — the step from working on one computer to working on many computers — is another example of a programming language making a context implicit; since the specifics of the computer that a program's code is run on is abstracted away from the program but can still influence evaluation, the

[5]The languages of these kinds of programs are usually referred to as **machine codes** because, in the modern era of computing, they are not meant to be written or read by programmers (or any humans for that matter).

specifics of the computer are a part of the program's implicit context.

However practically beneficial this abstraction using implicit contexts is, it does not come for free. In the definition of 𝔸, there appear to be no implicit contexts at all — every term can be fully evaluated with only rules defined by the language[6]. In fact, 𝔸 is defined such that the execution of its programs is completely independent from the physical state of the world. This is very useful for formal analyses, since 𝔸 programs exist purely as mathematical structures. However, this aspect also yields a language that is inert relative to the physical world; 𝔸 cannot *do* anything that requires reference to an implicit context. For example, it is definitionally impossible to write the standard "hello world"[7]. program in 𝔸, since printing to the screen requires the implicit context of all the mechanisms involved in the action of making a few pixels change color. As can be seen from this lacking, there are a lot of things that people, including professional programmers, would like to be able to do in a language that something as purely-mathematical as 𝔸 is unable to facilitate.

Throughout much of the history of computer engineering, this perspective has reigned so dominant that the thought of using something like 𝔸 (or an untyped but similarly-pure language, Lisp) for anything useful was rarely entertained.

> [**TODO**] talk about how object-oriented style is a particularly relevant branch of imperative programming in terms of dealing with implicit context. Objects are like little user-defined implicit contexts (stateful).

Today, most of the most popular programming languages are imperative: Java, C, Python, C++, C#, JavaScript, Go, R[8]. So why has this work introduced 𝔸 in the first place?

From however unpopular origins, the benefits of some features from more purely-mathematical, declarative languages have started making their way into popular programming language design. These adoptions have primarily taken two forms: (1) Convenient semantic structures, such as *λ-expressions* (a.k.a. *anonymous functions*) which were introduced into standard Java, JavaScript, and C++ in the early 2010's. (2) Abstraction-friendly and safer type systems, such as *generics* which were introduced into Java and C++[9] in the early 2000's, and the development of Scala, a functional language that compiles to Java virtual machine byte code. It turns out that having more easily-analyzable code is very useful for building effective, large-scale software. There is a lot to discuss in this direction, especially the topic of formal specification and verification, but that is beyond the scope of this work.

[6] It could be argued that there are in fact implicit contexts involved in specific implementations of 𝔸, since those implementations have to use computer memory to keep track of the data used in a 𝔸 program. However, this does in fact not count as an implicit context relative the *definition* of language A since

[7] This program that comes in many forms, but is used as a standard first program for learners of a new programming language. In general, executing a "hello world" program will print the string "hello world" to the console, screen, or some other visible output

[8] *10 Most Popular Programming Languages In 2020: Learn To Code.* https://fossbytes.com/most-popular-programming-languages/

[9] Generics are implemented in C++ using *templates*.

Though there are certain benefits to mathematically-inspired languages, this meshing of programming paradigms is still fiercely wresting with how to deal with implicit contexts. It is the following dilemma:

**The Dilemma of Implicit Contexts**

(I) Require fully explicit terms and reductions. This grants reasoning about programs to be fully formalized and abstracted from the annoyances of hardware and lower-level-implementations, but restricts such programs' use in applications that rely on implicit contexts.

(II) Allow implicit contexts that affect reductions. This grants many useful program applications and maintains some formal nature to the language's behavior, but reasoning about programs is now rife with considerations of implicit contexts.

The current state of affairs among popular programming languages is to choose horn II because of its pragmatic features, such as easy interaction with language-external peripheries, are so integral to software development. In fact, as the dilemma is presented it would seem that no real-world application could be feasibly written under horn I. The Dilemma of Implicit Contexts presents a good intuition for what is at stake in each case, but it relies on the concept of "implicit contexts" for meaning — a concept that is very fluidly used in programming language design. So, in the interest of specifying a place for the desired formalism of the horn I, the next section presents the concept of "effects."

### 2.2.1 Effects

Effects are the interfaces to implicit contexts that exist syntactically as well as semantically in programming languages. In this way, implicit contexts can also be embedded among syntactical contexts (e.g. scopes) as well as non-syntactical contexts (e.g. execution state); what is *implicit* from the point of view of a piece of code depends both on its context in the program in which its embedded and its context in the execution state in which it's evaluated. For a given expression with a given scope, the *explicit* factors (i.e. factors that can be referenced from within the expression) that affect the expression's evaluation are called *in-scope*, and the *implicit* factors (i.e. factors that cannot be referenced from within the expression) that affect the expression's evaluation are called *out-of-scope*.

> For an expression, a computational **effect** is a factor that affects the expression's evaluation but is outside the expression's normal scope.

An expression's *normal scope* is defined by the expression's used language, where "normal" indicates that some languages may have exceptional, abnormal scope-rules for certain circumstances that still yield effects. For example, many languages have *built-in* functions that, while appearing to be normal functions written in the language, actually directly interface with lower-level code not expressible in the language. Some of these built-in functions

has exceptional rules that allow it to refer to special factors that are indeed outside of a normal expression's scope. For example, a built-in function common to most programming languages is the *print* function, which sends some data to a peripheral output (such as you're computer's screen). Since *print* is an interface to the implicit context of the mechanics behind handling that peripheral, it is effectual.

A program is **pure** if it has no effects.

An expression is **impure** if it has effects.

There are a variety of effects that are available in almost every programming language. Interfaces to these effects are usually implemented as impure built-in functions (e.g. *print*) that are, within the language, indistinguishable from pure functions. [10] I shall use four common effects as running examples throughout the rest of this work: *input/ouput*, *mutability*, *exception*, and *nondeterminism*. They are detailed in the following paragraphs.

**2.2.1.0.1 Input/Output (I/O).** Any effect that involves interfacing with a context peripheral to the program's logic is called an I/O effect. Typically, these effects involve querying for information (input) or sending information (output). Examples include: printing to the console, accepting user input, reading or writing specific memory, displaying graphics, calling foreign (i.e. language-external) functions, reading from and writing to files.

**2.2.1.0.2 Mutability.** This is the effect of maintaining execution state that keeps track of variables' values over time and allows them to be changed. A languages's interface to mutability has three components: creating new variables, getting the value of a variable, and setting the new value of a variable. In this way, a variable can be thought of as a reference to some memory that stores a value, where the memory can be read for its current value or set to a different value without changing the reference itself. The setting of a variable's stored value to a new value is called **mutating** the variable.

**2.2.1.0.3 Exception.** This is the effect of an expression yielding an invalid value according to its specification. A language's interface to exception has two components: throwing an exception and catching an exception. Throwing an exception indicates that the expected sort of result of an expression would be invalid, according to the specification of that expression. Catching an exception in an expression is the structure of anticipating a throw of the exception during the evaluation of an expression, and having a pre-defined way of responding to such a throw should it occur. Examples of instances where exceptions are thrown: division by `0`, out-of-bounds indexing of an array, the head element of an empty list, calling a method of a null object-instance.

[10] These interfaces can sometimes be non-obvious. For example, in most declarative languages, mutability is so fundamental that it is not syntactically distinguished from normal assignment — almost all names are mutable saving marked exceptions.

**2.2.1.0.4 Nondeterminism.** This is the effect of an expression having multiple ways to evaluate. Three main strategies of implementing nondeterminism are (1) a deterministic model that accumulates and propagates all possible results, (2) a deterministic model that uses an initialized seed to produce nondeterminism for an unfixed seed, and (3) using I/O to produce a nondeterministic result. Examples of nondeterminism are: flipping a coin, generating a random number, selecting from a random distribution, probabilistic programming in general.

In addition to the above effects, **sequencing** is a control-flow structure specifically for effects. It is the effect of dictating the order in which other effects are performed. Sequencing is a very simple effect and is useful especially in declarative programing languages that aren't structured to be executed in a particular order. In imperative programming languages, sequencing is automatic since explicit execution order is required inherently.

However, many expressions both in declarative and imperative languages do not have effects. For example, suppose we have a function `max` that takes two integers as input and returns that larger one. If `max` is designed and implemented exactly to this specification, then `max` has no effects; the evaluation of `max` given some integers is not influenced by any factors outside of its normal scope, since the result is completely determined by its two explicit integer parameters.[11]

So now the sides of the Dilemma of Implicit Contexts can be considered more specifically given the concept of "effect". As is made clear by definition **??**, there is an easy method for transforming an impure expression into a pure expression: add the implicit factors depended upon by the expression's effects to the program's scope — now they are explicit and thus no longer effects! This reveals a more formal way of reasoning about effects in the terminology of programming language design. Still there is a dilemma of course, so using this intuition we can reformulate the Dilemma of Implicit Contexts in the terminology of effects:

### The Dilemma of Purity and Effect

(1) Require purity. This grants reasoning to depend only on normal scopes, which is very localized and explicit. This sacrifices convenience in and sometimes the possiblity of writing many pragmatic applciations.

(2) Allow both purity and impurity. This grants many useful applications where the behavior of the programs depends on factors not entirely encapsulated by the program's normal scope. This sacrifices the benefits of explicit and localized reasoning that depends only on normall scopes.

[TODO] transition from this dilemma to the motivation for defining 𝕀𝔹

[11] TODO: talk about how adding integers doesn't exactly work, but can be made to work with certain caveats, but its complicated (wrapping around overflows or erroring on overflows.)

## 2.3 Language 𝔹

> [**TODO**] mention that 𝔹 is supposed to stand as an implementation of effects in general, not just these particular effects. Pose a goal for the invention of 𝔹, to measure it by.

Language 𝔹 is a first example of implementing effects by extending 𝔸. 𝔹's approach to effects is inspired by Standard ML, which is commonly used in teaching programming language theory. This strategy is to introduce implicit execution contexts that primitive 𝔹 terms interact with to produce effects. So first, these primitive terms are introduced in the following subsections.

### 2.3.1 Primitives in 𝔹

#### 2.3.1.1 Mutability

A simple way of introducing mutability to a declarative language like 𝔸 is to posit a primitive type `mutable`, which is parametrized by a type $\alpha$, as the type of mutable $\alpha$-terms.

The term `initialize` is the single constructor for terms of type `mutable`. It takes an initial value $a{:}\alpha$ and evaluates to a new `mutable` $\alpha$ that stores $a$. The term `get` gets the value stored by a given mutable. The term `set` sets the value stored by a given mutable to a new given value.

Listing 2.1: Primitives for mutablility

```
primitive type mutable : Type → Type.

primitive term initialize (α : Type) : α → mutable α.
primitive term get (α : Type) : mutable α → α.
primitive term set (α : Type) : mutable α → α → unit.
```

Listing 2.2: Notations for mutability.

```
&(«term» : «type»)    ::=   initialize «type» «term»

!(«term» : «type»)    ::=   get «type» «term»

«term»₁ ← («term»₂ : «type»)    ::=   set «type» «term»₁ «term»₂
```

#### 2.3.1.2 Exceptions

When a program reaches a step where a lower-level procedure fails or no steps further steps forward are defined, the program fails. A general way of describing this phenomenon is *partiality* — programs that are undefined on certain inputs (in certain contexts, if effectual). For example, division is partial on the domain of integers, since if the second input is 0 then division is undefined.

One common way of anticipating partiality is to introduce *exceptions*, which are specially-defined ways for an expression to evaluate when it is not valid according to its specification. The immediate issue with extending 𝔸 with naive exceptions is that it requires exceptions to be of the same type as the expected result. Schematically, our division function example looks like this:

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

Suppose that we do not have exceptions, like in 𝔸. Then in the definition of `division`, the result of whatever is implemented in place of *"throw exception"* must yield an integer. This amounts to, however, the delegating a somewhat-arbitrary result in place of undefined results[12].

A simple declarative-friendly way to allow implicitly-exceptional results is to introduce a term that uses an exception instance to produce an *excepted* term of any type. We posit a Type, `Exception` $\alpha$, where terms of type `exception-of` $\alpha$ are each a label for exceptions parametrized by $\alpha$. Such exception-label should only be constructed primitively, so for the creating of new terms of type `exception-of` $a$ we introduce a new declaration: `exception` «*exception-name*» `of` «*type*».

Instances of `exception-of` $\alpha$ (which must be introduced primitively) are specific exceptions that are parametrized by a term of type $\alpha$. This allows exceptions to store some data, perhaps about the input that caused their throw. The term `throw` throws an exception, requiring its $\alpha$-input, and evaluating to any $\beta$-result. The term `catching` takes a continuation of type `(exception-of` $\alpha \to \alpha \to \beta$`)` for handling any $\alpha$-exceptions while evaluating a given term of type $\beta$. If an exception is thrown while evaluating a $b : \beta$, then `catching` results in the continuation, supplied with the specific thrown exception and its argument, instead of continuing to evaluate $b$. Note that a **continuation** is an expression that is triggered to "continue on" the evaluation of some larger expression when one of its sub-expression "escapes out."

[12]Though I deride it here, sometimes this strategy is actually used in practice. In Python 3.6, the string class's `find` method returns either the index of an input string in the string instance if the string is found, or a `-1` if the the input string is not found. However, the list class's `index` method yields a runtime error when the input is not found in the list instance.

Π 2.1: Exception syntax for IB

| metavariable | constructor | name |
|---|---|---|
| «*declaration*» | `exception` «*exception-name*» `of` «*type*»`.` | exception declaration |
| «*type*» | `exception-of` «*type*» | exception type |

Listing 2.3: Definitions for exception

```
primitive term throw (α β : Type) : exception-of α → α → β.

primitive term catching (α β : Type)
  : (exception-of α → α → β) → β → β.
```

Listing 2.4: Notation for exception

```
catch { «exception-name» («term-param»₁:«type»₁) ⇒ («term»₂:«type»₂) }
  in «term»₃

::=

catching «type»₁ «type»₂ «exception-name» ((«term-param»₁:«type»₁) ⇒ «term»₂)
  «term»₃
```

So in IB with this exception framework, we can schematically write the division function via the following.

```
1 exception division-by-0 of integer.
2
3 term division (x y) : float
4   := if y == 0
5        then throw division-by-0 x
6        else i/j
```

First, line 1 declares a new term `division-by-0 : exception-of integer`. Then, the definition of `division` on line 5 can evaluate to `throw division-by-0 x` to indicate that an attempt to divide by `0` has occurred.

Notice how the term `throw division-by-0 x` on line 5 is typed as an `integer`, yet it is not an integer. The throwing of an exception depends on the implicit context of some

surrounding `catch` structure in order to behave as if `division` meets the specification of returning a quotient. In the case that there is no surrounding `catch`, an implicit default catching mechanism will be triggered — usually a language-external error.

#### 2.3.1.3 I/O

I/O is a relatively generic effect since it offloads most of its details to an external interface. In other words: in order to capture all the capabilities of this interface, the representation of I/O within our language must be very general. As an easy setup, we shall use an I/O interface that just deals with `string`s. However, it is easy to imagine other specific I/O functions that would work in a similar manner (e.g. `input-integer`, `input-time`, `output-image`, etc.).

Listing 2.5: Definitions for I/O

```
primitive term input : unit → string.
primitive term output : string → unit.
```

The term `input` receives a string from the I/O interface, and the term `output` sends a string to the I/O interface. Note that `input` is of type `unit → string` rather than just `string`. This is because if we had `input : string` then it would refer to just one particular string value rather than possibly being reevaluated (receiving another input via I/O) by using a dummy parameter `unit`.

#### 2.3.1.4 Nondeterminism

In 𝔹, nondeterminism is simply implemented via a special interface similar to how I/O is implemented. The following functions define that interface to the language.

Listing 2.6: Primitives for nondeterminism

```
primitive term random-float : unit → float.

term random-range-float
      (min : float) (max : float) (_ : unit) : float
  := min + (max - min) * (random-float •).

term random-range-integer
      (min : integer) (max : integer) (_ : unit) : integer
  := min + (max - min) * floor (random-float •).

term random-bool : unit → boolean
  := (x ⇒ x < 0.5) ∘ random-float.
```

#### 2.3.1.5 Sequencing and Binding

> [TODO] is there a problem here in that the right side of the sequence will evaluate before the left side? or do I just ignore this for now and leave it as a special reduction rule

Since effects are context-sensitive in a way visible from within 𝔸, we'd like 𝕀𝔹 to allow express, explicitly, the order in which effects are to be performed. Such an expression is an application of the *sequencing* effect.

A primitive term allows sequences to be written, and 𝕀𝔹's reduction rules mirror them by reducing terms in the correct order.

Listing 2.7: Primitive for sequencing

```
primitive term sequence (α β : Type) : α → β → β.
```

Listing 2.8: 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)`

Additionally, it is convenient to introduce bound terms, similarly to the `let` binding structure, within `do` blocks. However, using the normal `let` structure is cumbersome, so we introduce a new notation for achieving this aesthetically:

Listing 2.9: Notation for sequenced binding

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

### 2.3.2 Reduction Rules for 𝕀𝔹

> [TODO] formally explain how reduction contexts work. Δ generalizes all reduction contexts.

[TODO] formally explain how reduction contexts work ($\mathcal{S}, \ldots \parallel a$). Should that go in this section, or the definition of $\mathbb{A}$? If I put it in $\mathbb{A}$, that would make defining let expressions much easier.

[TODO] explain how $\Delta$ is handled when not included in inference rule (stay same)

Π 2.2: Reduction in $\mathbb{B}$

$$\textsc{Simplify}\quad \frac{\Delta \parallel b \twoheadrightarrow \Delta' \parallel b'}{\Delta \parallel a\ b \twoheadrightarrow \Delta' \parallel a\ b'}$$

$$\textsc{Simplify}\quad \frac{\Delta \parallel a \twoheadrightarrow \Delta' \parallel a' \qquad \text{value } v}{\Delta \parallel a\ v \twoheadrightarrow \Delta' \parallel a'\ v}$$

#### 2.3.2.1 Reduction Rules for Mutability

[TODO] explain how $\mathcal{S}$ works, with the whole $\mathcal{S}[\![i \mapsto v]\!]$ deal

[TODO] Define what $\mathcal{S}$ looks like mathematically (a mapping, where indices form a set that can be looked at).

[TODO] define newUID(S)

Π 2.3: Reduction in $\mathbb{B}$: Mutability

$$\textsc{Initialize}\quad \frac{\text{value } v \qquad (\mathcal{S}', i) \coloneqq \mathsf{new}(\mathcal{S})}{\mathcal{S} \parallel \&v \twoheadrightarrow \mathcal{S}[\![i \mapsto v]\!] \parallel i}$$

$$\textsc{Get}\quad \mathcal{S}[\![i \mapsto v]\!] \parallel !i \twoheadrightarrow \mathcal{S}[\![i \mapsto v]\!] \parallel v$$

$$\textsc{Set}\quad \frac{\text{value } v'}{\mathcal{S}[\![i \mapsto v]\!] \parallel i \leftarrow v' \twoheadrightarrow \mathcal{S}[\![i \mapsto v']\!] \parallel \bullet}$$

[TODO] english explanation

#### 2.3.2.2 Reduction Rules for Exception

[TODO] introduce idea of **reduction context**. There can be many separate reduction contexts (symbolized by capital script letters), and the total context $\Delta$ symbolizes all other reduction contexts not specifically referenced.

[TODO] explain how $\mathcal{E}$ works, how to interpret the $\mathcal{E}\{\emptyset\}$ and $\mathcal{E}\{\varepsilon, x\}$ things

[TODO] Note the there must be a priority order to these reduction rules, because it matters which are applied in what order (as opposed to other rules).

[TODO] use append operation for adding things to top of stack

Π 2.4: Reduction in IB: Exceptions

$$\textsc{Throw}\quad \frac{\text{value } x}{\mathcal{E}\{\emptyset\} \parallel \texttt{throw}\ \varepsilon\ x \twoheadrightarrow \mathcal{E}\{\varepsilon, x\} \parallel \text{♪}}$$

$$\textsc{Catch}\quad \mathcal{E}\{\varepsilon, x\} \parallel \texttt{catch}\ \{\ \varepsilon\ a \Rightarrow b\ \}\ \texttt{in}\ \text{♪} \twoheadrightarrow \mathcal{E}\{\emptyset\} \parallel (a \Rightarrow b)\ x$$

$$\textsc{Catch-Pass}\quad \frac{\text{value } v}{\mathcal{E}\{\emptyset\} \parallel \texttt{catch}\ \{\ \varepsilon\ a \Rightarrow b\ \}\ \texttt{in}\ v \twoheadrightarrow \mathcal{E}\{\emptyset\} \parallel v}$$

$$\textsc{Raise}\quad \mathcal{E}\{\varepsilon, x\} \parallel \texttt{a}\ \text{♪} \twoheadrightarrow \mathcal{E}\{\varepsilon, x\} \parallel \text{♪}$$

$$\textsc{Raise}\quad \mathcal{E}\{\varepsilon, x\} \parallel \text{♪}\ \texttt{b} \twoheadrightarrow \mathcal{E}\{\varepsilon, x\} \parallel \text{♪}$$

[TODO] english explanation

#### 2.3.2.3 Reduction Rules for I/O

Π 2.5: Reduction in IB: I/O

$$\textsc{Input}\quad \frac{\text{value } v}{\mathcal{IO} \parallel \texttt{input}\ v \twoheadrightarrow \mathcal{IO} \parallel \mathcal{IO}(\texttt{input}\ v)}$$

$$\textsc{Output}\quad \frac{\text{value } v}{\mathcal{IO} \parallel \texttt{output}\ v \twoheadrightarrow \mathcal{IO} \parallel \mathcal{IO}(\texttt{output}\ v)}$$

These rules interact with the I/O context, $\mathcal{IO}$, by using it as an interface to an external I/O-environment that handles the I/O effects. This organization makes semantically explicit the division between IB's model and an external world of effectual computations. For example, though $\mathcal{IO}$ is an interface to a stateful context, the state cannot be directly represented in IB's semantics; $\mathcal{IO}$ is a black box from the point of view within IB. An external implementation of $\mathcal{IO}$ that could be compatible with IB must satisfy the following specifications:

##### 2.3.2.3.1 Specification of $\mathcal{IO}$

- $\mathcal{IO}(\texttt{input}\ \bullet)$ resolves as a value of type `string`,

- 𝒹𝒪(output s) resolves as •.

I use the term "resolve" here to emphasize that 𝒹𝒪's capabilities are operating outside of the usual semantics of 𝕀𝔹 in order to evaluate.

[**TODO**] come up with better running I/O example?

At this point, we can express the familiar Hello World program.

Listing 2.10: Hello World

```
output "hello world"
```

But as far as the definition of 𝕀𝔹 is concerned, this term is treated just like any other term that evaluates to •. The implementation for 𝒹𝒪 used for running this program decides its effectual behavior (within the constraints of the specification of 𝒹𝒪 of course). An informal but satisfactory implementation is the following:

#### 2.3.2.3.2 Implementation 1 of 𝒹𝒪:

- 𝒹𝒪(input •):
  1. Prompt the console for user text input.
  2. Interpret the user text input as a string, then resolve as the string.
- 𝒹𝒪(output s):
  1. Write s to the console.
  2. Resolve as •.

As intended by 𝕀𝔹's design, this implementation will facilitate Hello World appropriately: the text "hello world" is displayed on the console. Beyond the requirements enumerated by the specification of 𝒹𝒪 however, 𝕀𝔹 does not guarantee anything about how 𝒹𝒪 behaves. Consider, for example, this alternative implementation:

#### 2.3.2.3.3 Implementation 2 of 𝒹𝒪:

- 𝒹𝒪(input •):
  1. Set the toaster periphery's mode to *currently toasting*.
  2. Resolve as a string representation of the toaster's current temperature.
- 𝒹𝒪(output s):

1. Interpret `s` as a ABH routing number, and route $1000 from the user's bank account to 123456789.
2. Set the toaster periphery's mode to *done toasting*.
3. Resolve as •.

This implementation does not seem to reflect the design of 𝔹, though unfortunately it is still compatible. In the way that 𝔹 is defined, it is difficult to formally specify any more detail about the behavior of I/O-like effects, since its semantics all but ignore the workings of 𝒽𝒪.

#### 2.3.2.4 Reduction Rules for Nondeterminism

Π 2.6: Reduction in 𝔹: Nondeterminism

RANDOM-FLOAT $\mathcal{R} \parallel \texttt{random-float}\ \bullet \twoheadrightarrow \mathcal{R} \parallel \mathcal{R}(\texttt{random-float}\ \bullet)$

[**TODO**] English explanation

##### 2.3.2.4.1 Specification of $\mathcal{R}$

- $\mathcal{R}(\texttt{random-float}\ \bullet)$ resolves to a value of type `float` that is no less than `0` and less than `1`.

#### 2.3.2.5 Reduction Rules for Sequences

Π 2.7: Reduction in 𝔹: Sequencing

$$\textsc{Sequence}\ \frac{\text{value } v}{v\ ;\ a \twoheadrightarrow a}$$

[**TODO**] English explanation

## 2.4 𝔹 in Action

[**TODO**] introduce some running examples for each of the effects

##### 2.4.0.0.1 Mutability.

##### 2.4.0.0.2 Exception.

##### 2.4.0.0.3 IO.

**2.4.0.0.4 Nondeterminism.** (Flipping a coin.)

## 2.5 Motivations

This chapter has introduced the concept of effects in programming languages, and presented IB as a simple way to extend a basic lambda calculus, A\, with a sample of effects. For each effect, the implementation strategy used is to introduce a facilitating reduction context: $\mathcal{S}, \mathcal{E}, \mathcal{IO}, \mathcal{R}$. Each these reduction contexts are examples of implicit contexts since they are not directly able to be referenced from within IB— they can only be interacted with via the primitive terms introduced for each effect.

Among these reduction contexts, there are two groups that naturally emerge: (1) $\mathcal{S}$ and $\mathcal{E}$, and (2) $\mathcal{IO}$ and $\mathcal{R}$. Considering group (1), $\mathcal{S}$ and $\mathcal{E}$ are treated purely-mathematical structures uniquely defined (up to affecting reduction behavior) by their specification. Call these sorts of effects **internal effects**, as are fully specified by the language-internal semantics. The context $\mathcal{S}$ is a mapping between unique identifiers and IB-values, and the context $\mathcal{E}$ is either empty or contains a pair of an IB-exception-name and a IB-value. While these reduction contexts may be implemented in a variety of ways while still meeting IB's specification for them, each of these implementations will have the same behavior from the point of view of reduction in IB.

Considering group (2), $\mathcal{IO}$ and $\mathcal{R}$ are treated as interfaces to a vaguely-specified external context that is implicit relative to the reduction rules. Call these sorts of effects **external effects**, as they are not fully specified by the language-internal semantics and must rely on language-external implementation. The context $\mathcal{IO}$ is a black-box interface that can either get input or receive output, and the context $\mathcal{R}$ is a black-box interface that can produce a random float.

A reason external effects are introduced to IB this way is because it is infeasible to introduce an entire model of their IB-implicit behaviors. For example, the `print s` capability of $\mathcal{IO}$ could involve running some low-level code to send data to a peripheral console, the details of which could not be written in IB. Internal effects, on the other hand, are easy to introduce completely-explicitly since its effects only involve simple structures that are easily and uniquely modeled as the mathematical structures described before.

While one would like to imagine that $\mathcal{IO}$ behaves in an intuitive way, the example of Implementation 2 of $\mathcal{IO}$ demonstrates that $\mathcal{IO}$ can behave very unexpectedly while still meeting IB's specification for it. The same applies to $\mathcal{R}$. This situation arises because, as a black-box, these contexts are hiding a lot of implicit activity that is ultimately relevant for their influence on reduction. So, differences in implementation, which govern the implicit activity, *are* relevant to reduction.

This is clearly a serious complication for any formal analysis of programs with $\mathcal{IO}$ or $\mathcal{R}$ effects in them. And, even if the top-level expression does not appear to have any such effects in it, it could be that names it references, which are defined somewhere else far away, could have such effects. It seems like almost anything could happen, but IB treats expressions with these effects exactly the same as expressions that don't!

In the following chapters, we shall consider a few alternative ways to introduce effects to 𝔸. The goal in doing so it to find strategies for representing effects that inherit the most advantage from the declarative style as well as sacrifice the fewest capabilities of the imperative style. Learning from this chapter's consideration of 𝔹, there are a couple inspirations for such improvement to take away.

**2.5.0.0.1 Generalized contexts for effects.** In 𝔹, each effect has a unique reduction context. So, in order to facilitate more effects, the reduction rules of the language must be changed. There are are many more useful effects than the examples introduced in this chapter, so this requirement proves especially cumbersome to the goal of posing 𝔹 as an implementation of effects *in general*. This observation begs for an abstract structure for effects in general, which could be handled in a language's reduction rules and instantiated as particular effects in code.

**2.5.0.0.2 Explicitly-typed effects.** In 𝔹, impure expressions are indistinguishable from pure expressions before reduction. Because of this is difficult for a programmer to identify the purity of parts of a program, which is useful for the formal analysis of expected behavior.[13] A common way of addressing this is to introduce a typing-structure that indicates when a value is produced effectually. Such a typing-structure ensures that the purity of expressions is handled explicitly within code, though it also introduces an extra layer of complexity.

**2.5.0.0.3 Programmable internal effects.** In 𝔹, internal and external effects are introduced in the same way — appealing to a reduction context. However, since internal effects can be fully specified language-internally, it seems possible to have a syntax structure for instantiating internal effects rather than relying on special reduction rules. In this way, a programmer could implement their own internal effects in code. Note that under this extension, expressions with internal effects and expressions with external effects need not behave differently at compile-time.

In the next chapter, we shall consider a new structure for modeling effects, the *monad*.

[13] A pragmatic result of this design desicion is to promote programs to segregate their code by purity, as making sure that types match between pure and impure expressions is annoying to handle when it is intermingled. This organization by purity presents opportunities for formal analyses such as verification (pure code can be verified at compile-time [TODO: citation]), modularity (pure code can be used anywhere and have the same behavior, impure code can be sometimes be combined [TODO: citation]), and optimization (impure activities that take more resources to do sporadically can be reorganized around pure code to improve efficiency, without changing behavior [TODO: citation]). These considerations are beyond the scope of this work, but provide a context for why these formal analyzability is desirable.

For example, suppose we are programming a chat bot. The chat bot should recieve input from the user (an impure activity), then run some computation with the input (a pure activity), and then finally display a result to the user (another impure acitivty). This program should natually be divided up into three functions, `get-user-input`, `compute-output`, and `display-output`, where the first and third are impure and the second is strictly pure.

Monadic effects will address these inspirations and, in doing so, refine our goals for implementing effects.

# K 3

# Monadic Effects

[**TODO**] make order of capabilities: lifting, mapping, binding

## 3.1 Introduction to Monads

[**TODO**] introduce in programmer-friendly way first, mention *category* only for like a sentence if at all. give examples of writing code in Java/C/etc. and how we want to be able to do something similar but in a functional way

[**TODO**] talk about terminology of "compution" and "results". the computation is in reference to monads and hteir internal context, and result is in reference to the pure part that the computation results in

[**TODO**] describe the motivation fo capabilities, that we want to extract the essential aspects to beign a monad. Also give preview for how the chapter is organized. Relate the LangB, where there is a left side (implicit context) and right side (explicit context).

The concept of *monad*, which originates in category theory, turns out to be a very convenient structure for formalizing implicit contexts within functional programming languages. While at the top level, everything written in a program is explicit, monads create a certain kind of internal environment for which the structure of a monad is rendered *relatively implicit* from "within" the monad; the structure of a monad is a *relatively implicit context*, relative to terms interfacing with the monad. This section will follow a programmer-friendly introduction to monads and how they can be used to implement effects in a new extension to 𝔸\.

[**TODO**] define capability, and how it will be used to defined classes of types e.g. monad.

A **capability** of a type is a term with a signature in which that type appears.

[TODO] describe how there are a few essential **capabilities** for saying something is a monad. In the next section we will discover what those are by considering a particular monad.

### 3.1.1 The Mutablility Monad

The mutability effect is particularly general, so we shall use it as a running example in introducing the details motivating and specifying monad. Mutability was implemented by 𝔹 by the introduction of a globally-accessible, store of mutable variables. This implementation required, however, several new syntactic structures and reduction rules. Is there a way to implement something similar without adding so many extensions of 𝔸?

[TODO] introduce terminology: $\sigma$-stateful $\alpha$-computation.

[TODO] decide and refactor for terminology: what exaclty does "computation" mean in this context? I'm thinking of it as an impure expression. But its a general term and probably not so useful to be confusing about here.

It turns out there is — with a certain tradeoff. Since 𝔸 is pure, such an implementation cannot provide true mutability. However, we can model mutability in 𝔸 as a function from the initial state to the modified state. A $\sigma$-stateful $\alpha$-computation is a stateful computation where the state is of type $\sigma$ and the result is of type $\alpha$. To describe the type of such computations purely, consider the following:

```
type mutable (σ a : Type) : Type := σ → σ × α.
```

Relating to the description of the mutability effect, `mutable σ α` is the type of functions from an initial $\sigma$-state to a pair of the affected (i.e. possibly-mutated) $\sigma$-state and the $\alpha$-result. So if given a term $m$ `: mutable σ α`, one can purely compute the affected state and result by providing $m$ with an initial state.

To truly be an effect, there needs to be an internal context in which mutability is implicit. So let us see how we can construct terms that work with `mutable`. In 𝔹's implementation of this effect, it posited two primitive terms: `get` and `set`. Using `mutable` we can define these terms in 𝔸 as:

```
term get (σ : Type)
  : mutable σ σ
  := s ⇒ (s, s).

term set (σ : Type)
  : σ → mutable σ unit
  := s' ⇒ (s ⇒ (s', •)).
```

Observe that `get` is a $\sigma$-stateful $\sigma$-computation that does not modify the state and results in the the current value of the state. And, `set` is a $\sigma$-stateful computation that replaces the state with a given $s' : \sigma$ and results in $\bullet$. In these ways, using `get` and `set` in $\mathbb{A}$ fills exactly the same role as a simple $\mathbb{B}$ mutable effect where $\sigma$ is a type corresponding to a store of values (perhaps a named product).

> [**TODO**] expand on this, show example of using $\mathbb{A}$ and $\mathbb{B}$ side-by-side (or maybe do this after the next part about sequencing etc.)

**3.1.1.0.1 Sequencing.** Since the mutable effect is in fact an effect, we should also be able to *sequence* stateful computations to produce one big stateful computation that does performs the computations in sequence. It is sufficient to define the **sequence** of just two effects, since any number of effects can be sequenced one step at a time. So, given two $\sigma$-computations $m$ `: mutable` $\sigma$ $\alpha$ and $m'$ `: mutable` $\sigma$ $\beta$ the sequenced $\sigma$-computation should first compute the $m$-affected state and then pass it to $m'$.

```
term mutable-sequence (σ α β : Type)
  : mutable σ α → mutable σ β → mutable σ β
  := m m' ⇒
       s ⇒ let (s', _) := m s in m' s'.
```

**3.1.1.0.2 Binding.** However, in this form it becomes clear that `mutable-sequence` throws away some information — the result of the first stateful computation. To avoid this amounts to allowing $m'$ to reference $m$'s result, which can be introduced by using the parameter $fm$ `: α → mutable` $\sigma$ $\beta$ in place fo $m'$. $fm$ is named such because it is a function (hence the $f$) to a monad term (hence the $m$). A sequence that allows this is called a *monadic binding-sequence*, or just **monadic bind** (or just **bind**), since it sequences $m, fm$ and also passes the result of $m$ to $fm$ (i.e. binding the result of $m$ to the first parameter of $fm$).

```
term mutable-bind (σ α β : Type)
  : mutable σ α → (α → mutable σ β) → mutable σ β
  := m fm ⇒
       s ⇒ let (s', a) := m s in fm a s'.
```

Additionally, one may notice that one of `mutable-sequence` and `mutable-bind` is superfluous i.e. can be defined in terms of the other. Consider the following re-definition of `sequence`:

```
term mutable-sequence (σ α β : Type)
  : mutable σ α → mutable σ β → mutable σ β
  := m m' ⇒ mutable-bind m (_ ⇒ m').
```

This construction demonstrates how `mutable-sequence` can be thought of as a sort of trivial bind, where the bound result of $m$ is ignored by $m'$.

> [**TODO**] take some examples of B code and write in terms of this formulation, side-by-side

So far, we have defined all of the state-relevant operations needed to express stateful computations in 𝔸\. The key difference between 𝔹 and our new 𝔸\-implementation of the mutable effect is that 𝔹 treats stateful computations just like any other kind of pure computation, whereas 𝔸\ "wraps" $\sigma$-stateful $\alpha$-computations using the type `mutable σ α` rather than just an "unwrapped" type $\alpha$. In this formulation, the stateful aspect of a computation must be handled in some way to extract the result. Otherwise, a monadic bind must be used to access the result, which propagates the `mutable` wrapper.

However, what is still missing any ways to trivially treat pure values as computations and to interact with a result without accessing the computations implicit context. There are two kinds of such embeddings: *lifting* and *mapping*.

**3.1.1.0.3 Lifting.** A (relatively[1]) pure value can be *lifted* to a stateful computation by having it be the result of the computation and not interact with the state. Suppose we would like to write an impure function `reset` that sets the integer-state to `0` and results in the previous state value — the function would have type `mutable integer integer`. With just `get`, `set`, and `bind`, there is no way to write this function as there is no way to combine them in such a way that sets the state but results in something other than •. The missing expression we need is a function that results in a value without touching the state at all. Call this function `mutable-lift`, which should have the type `α → mutable σ α`.

```
term mutable-lift (α : Type) : mutable σ α
  := a ⇒
       s ⇒ (s, a)
```

So, we can construct `reset` to first store the old state in a name `old`, set the state to `0`, and finally use `lift` to result in `old`. Given this description, the construction of `reset`, using `lift`, is the following:

```
term reset : mutable integer integer
  := mutable-bind get
       (old ⇒ mutable-sequence
```

[1] Relative to the considered monad. There is no restriction that the lifted value is not a monadic term itself, which yields nested monads. These nested monads may be different, but if they are all of the same monad then they can be collapsed via the function `join` (see Prelude for ℂ, Appendix A).

```
        (set 0)
        (lift old)).
```

**3.1.1.0.4 Mapping.** A relatively pure function can be be *mapped*[2] over stateful computation results by applying it to the result of the computation and then lifting. In other words, to lift[3] a function of type `α → β` to a function of type `mutable σ α → mutable σ β`. Call this function `mutable-map`.

```
term mutable-map (σ α β : kind) : (α → β) → mutable σ α → mutable σ β
  := f ⇒ mutable-bind get (mutable-lift ∘ f).
```

As seen above, `mutable-map` arises very straightforwardly from `mutable-lift`. However, this is not the case in general. Each monad instance will have a particular set of implementations for lifting and mapping, as well as binding.

> [TODO] example? might not really be able to, since goes into too much later stuff

## 3.1.2 Formalizion of Monad

### 3.1.2.1 Definition of Monad

In section 3.1.1, we implemented a variety of terms specifically for the mutability monad. To summarize, they were: `get`, `set`, `mutable-sequence`, `mutable-bind`, `mutable-lift`, and `mutable-map`. A selection of these are essential to being a monad, but as they were implemented they will only work with one particular monad. We would like to extract the essentially-monadic capabilities that some of these terms provide for the mutability monad, and describe them generally as applying to all monad instances.

Firstly, the terms `get` and `set` were exclusively implemented for the structure of `mutable`. So they must not be monad-essential, since to be a monad must not depend on the structure of `mutable`. Secondly, the term `mutable-sequence` provides the monad-essential capability of sequencing, but was later discovered to be expressible in terms of `mutable-bind` without reference to the particular structure of `mutable`. Both sequencing and binding are monad essential, but to be minimal we need only explicitly require binding.

[2]The term *mapping* is very much overused between computer science and mathematics terminology. Here, to **map** a function over a data structure is to apply the function to the contents of the data structure (in some way relevant to the data structure). For example, mapping a function $f$ over a list $[1, 2, 3]$ simplifies to $[f\ 1, f\ 2, f\ 3]$.

[3]I use the term "lift" both for the previous paragraph on *lifting* and this paragraph on *mapping*. The idea behind this more general concept of "lift" is the embedding of a simpler term into a more complex term in some trivial way. In this case, lifting values to stateful computation results is called *lifting*, and lifting functions to functions between stateful computation results is called *mapping*.

Thirdly, the terms `mutable-lift` and `mutable-map` provide the monad–essential capabilities of lifting and mapping. Though `mutable-map` was implemented using `mutable-lift`, this implementation required reference to the particular structure of `mutable` so this does not collapse lifting and mapping in the way that sequencing and binding collapsed.

So, minimally, there are three essential capabilities required of a structure to be a monad: binding, lifting, and mapping. Since mapping is covered by being a functor, we can rephrase this as a definition of "monad":

> A **monad** is a functor with binding and lifting capabilities i.e. in a sequence of two monads the result of the first term can be bound in the second term, and relatively pure values can be lifted to monad terms.

However, within 𝔸 we currently have no type-oriented way to assert that a type $M$ is associated with the expected constructions for qualifying as a monad. How could the property of "being a monad" be represented in 𝔸?

## 3.2 Type-classes

It turns out that *type-classes* are a particularly clean way of representing that terms of particular types have certain properties. So before fully extending 𝔸 with monads, let us first take a detour on *type-classes*, of which monad is just one sort.

### 3.2.1 Concept of Classes

The concept of having a *class* of structures is used in many different forms among many different programming languages. In object-oriented programming languages, **object-classes** define "blueprints" for creating objects which are **object-instances** of the class. Such an object-class defines an *interface* that each instance of the class must implement. In this way, an object-class specifies an interface to a class of object-instances. We shall use Java to demonstrate the object-oriented implementation of classes, and the following is an example of a simple object-class that specifies an interface to lists of integers.

```
class IntegerList {
  public int head;
  public IntegerList tail;

  // nil, when given no arguments
  public IntegerList() {
    this.head = null;
    this.tail = null;
  }

  // cons, when given two arguments
  public IntegerList(int head, IntegerList tail) {
```

```
    this.head = head;
    this.tail = tail;
  }
}
```

Observe how `IntegerList` is structured similarly to `list` in 𝔸 (see Appendix A, Prelude for 𝔸), though `IntegerList` appears relatively clunky in Java.

In relating object-classes to 𝔸's framework, we use the following analogy: *object-instances are to terms as object-classes are to types*. For example, the object-instance `new IntegerList (1, IntegerList())` is to the object-class `IntegerList` as the term `[1]` is to the type `list integer`. This is all fine and intuitive as we are familiar with the term-type relationship from working with 𝔸 and strictly-typed languages in general. But what about classes of object-classes? In other words, how would we define a higher-order object-class-like that specifies behavior for a class of object-classes? Note that we ask this question in order to approach the question of representing classes of types in 𝔸.

In object-oriented programming, these higher-order structures are called abstract object-classes. An abstract object-class specifies the types of a selection of methods so, for an object-class to be an instance[4] of the abstract object-class So then, in order for an object-class to be an instance of an abstract object-class, the object-class must implement methods and fields of the correct names and types that are specified by the abstract object-class. This analogizes to the requirements, to provide certain capabilities, of a type in order to be a monad. For an example of an abstract object-class, the following defines an abstract class `Animal` as the class of object-classes that have two methods that take no arguments and return `String` called `eat` and `sleep`.

```
abstract class Animal {

  abstract public String eat();
  abstract public String sleep();

}
```

As is commonly known, both cat and dog are examples of species of animals, so we can create respective object-classes that are instances of the `Animal`, which is indicated by the `extends Animal` clause:

```
class Cat extends Animal {

  String name;
  public Cat(String name) { this.name = name; }

  public String eat()   { return this.name + " eats kibble."); }
```

[4] I am stretching terminology a little here. In normal object-oriented lingo, what I am calling an instance of an abstract object-class would rather be called a sub-class of the abstract object-class. Hence the Java code's use of `extends` to instantiate an abtract object-class.

```
  public String sleep() { return this.name + " naps."); }
  public String hunt()  { return this.name + " hunts for mice."); }

}

class Dog extends Animal {

  String name;
  public Dog(String name) { this.name = name; }

  public String eat()   { return this.name + " eats steak."); }
  public String sleep() { return this.name + " sleeps wrestlessly."); }
  public String walk()  { return this.name + " is walked by human."); }

}
```

Note that while `Cat` and `Dog` do not share all the same methods and fields, the fact that they are both instances of `Animal` guarantees that they at least meet the specification defined by `Animal`. Since `Animal` abstracts over a class of object-classes, we can write functions that work on any object-instance of an object-class instance of `Animal`. In this example, one can write functions that can work with both `Cat` and `Dog` instances by only assuming an interface specified by `Animal`.

```
string simulate_Animal(Animal a, int steps) {
  String simulation = "";           // gathers a description of
                                    // the animal's behavior.
  for (int i = 0; i < 10; i += 1) { // loop for the number of steps.
    simulation += a.eat();          // each step, the animal eats
    simulation += a.sleep();        // and sleeps.
  }
  return simulation;
}
```

This example gives a simple demonstration of how the concept of classes can be introduced into programming semantics. Recall the analogy of object-instances, terms, object-classes, and types. Where do abstract object-classes fit into this analogy? In the next section, *type-classes* are introduced to the 𝔸 framework to parallel how abstract object-classes behave in object-oriented programming. So the analogy completes as *object-instances are to terms as object-classes are to types as abstract object-classes are to type-classes.*

### 3.2.2 Type-classes

A **type-class**, with a parameter type, is defined by a selection of capabilities that involve the parameter type. A capability takes the form of a name with a type, indicating that a

term with the name and of the type must be implemented for each instance. There are two parts to introducing classes in code: class definition and class instantiation.

The most straightforward representation of a type-class as a type is as an $n$-ary product type, with a component for each capability. For example, the following is a translation of the abstract object-class `Animal` into 𝔸\:

```
// type-class
type Animal (α : Type)
  := α → unit → unit // eat
  × α → unit → unit // sleep
```

A term of type `Animal α` for some type $\alpha$ is a sort of container (as a product) implementation for these capabilities defined for `Animal`. More generally, another way to think of this is that a term of type $C\ A$ for some type-class $C$ and type $A$ is a *proof* that $A$ is an instance of $C$. This is because the term itself contains the implementation of each of the capabilities required for type-class membership of the type. Going forward, this term for a type-class instance will be called `impl`, as the implementation of the type-class instance. To extract a particular capability, the $i$-th part of `impl` is taken where $i$ is the index of the capability in the product. We name the class `cat`, with instance parameter `α`, and name the class capabilities `name`, `eat`, and `sleep`.

```
type Animal (α : Type) : Type
  := (α → string)  // name
  × (α → string)  // eat
  × (α → string). // sleep

term name  (α : Type) (impl : Animal α) := part-1 impl.
term eat   (α : Type) (impl : Animal α) := part-2 impl.
term sleep (α : Type) (impl : Animal α) := part-3 impl.
```

Likewise instantiating the types `cat` and `dog` as instances of the class `Animal` is written as the following:

```
// definition of cat
type cat := string.
term cat-name  (c : cat) := c.
term cat-eat   (c : cat) := cat-name c ++ " eats kibble."
term cat-sleep (c : cat) := cat-name c ++ " naps."
term cat-hunt  (c : cat) := cat-name c ++ " hunts for mice."

// instantiate cat as an instance of class Animal
term Animal-cat : Animal cat := (cat-eat, cat-sleep).

// definition of dog
type dog := string.
term dog-name  (d : dog) := d.
```

```
term dog-eat   (d : dog) := dog-name d ++ " eats steak.".
term dog-sleep (d : dog) := dog-name d ++ " sleeps wrestlessly.".
term dog-walk  (d : dog) := dog-name d ++ " is walked by human.".

// instantiate dog as an instance of class Animal
term Animal-dog : Animal dog := (dog-eat, dog-sleep).
```

Given the type-class `Animal` with its instances `cat` and `dog`, we can write a translation of the Java function `simulate_Animal`:

```
term simulate_Animal (α : Type)
      (impl : Animal α) (a : α) (steps : natural)
  : string
  := (string-concat ∘ repeat steps)
      (eat impl a ++ sleep impl a).
```

#### 3.2.2.1 Notations for type-classes

Notice how in the previous section, the style of specification of `Animal`, instantiation of `Animal`, and requirement of the `impl` parameter to use capabilities of `Animal` are particularly clunky and unintuitive. They look more like a raw, underlying implementation full of details that don't relate simply to the idea of classes of types. It would be nice to not have to define the terms *animal*`-name`, *animal*`-eat`, etc. for each instance *animal* of `Animal`, since the general terms `eat`, `sleep` are already defined to work on all `Animal`s. There should be a more simple way of expressing type-class specifications and instantiations than requiring all this boilerplate. It turns out that there are some handy notations we can use to capture and simplify the expression of type-classes.

**3.2.2.1.1 Type-class Specification.** A type-class is defined by the specification is gives for its instances. The specification consists of a collection of capabilities which are the types of terms that each type-class instance must implement. Given this construction, it is possible to write a generalized term for each capability that works for any type-class instance. The following notation allows the concise specification of a type-class by just its capabilities, and additionally automatically generates the generalized terms for each capability.

Listing 3.1: Notation for type-class specification

```
class «class-name» («type-param»_* : «kind»_*) : «kind»_c
  { «term-name»_1 : «type»_1 ; ⋯ ; «term-name»_n : «type»_n }.

::=

type «class-name» («type-param»_* : «kind»_*) : «kind»_c
  := «type»_1 × ⋯ × «type»_n.
```

```
term «term-name»_1 («type-param»_*:«kind»_*) ⟦ («type-param»_i:«kind»_i) ⟧
       (impl : «class-name» «type-param»_*)
  : «type»_j
  := part-1 impl.
⋮
term «term-name»_n («type-param»_*:«kind»_*) ⟦ («type-param»_i:«kind»_i) ⟧
       (impl : «class-name» «type-param»_*)
  : «type»_n
  := part-n impl.
```

Note that is is a little cumbersome to have to provide the `impl` argument each time one of the generalized terms is used. To avoid this, we shall adopt the convention that the `impl` argument is passed implicitly if the type being used has previously been instantiated of the type-class.

**3.2.2.1.2 Type-class Instantiation.** Instantiating the a type $\alpha$ as an instance of a type-class $C$ requires implementing the terms specified by $C$ where $A$ is supplied as the argument to $C$'s first type parameter. The following notation conveniently names this implementation and makes explicit $C$'s intended names for each component.

Listing 3.2: Notation for type-class instantiation

```
instance ⟦ («type-param»_i:«kind»_i) ⟧ ⇒ «class-name» «type»_*
  { «term-name»_1:«type»_1 := «term»_1 ; ⋯ ; «term-name»_n:«type»_n := «term»_n }.

::=

term «class-name»-«type»
  : ⟦ («type-param»_i:«kind»_i) ⟧ ⇒ «class-name» «type»_*
  := («term»_1, …, «term»_n).
```

The notation expands to constructing a term with the name «*class-name*»-«*type*», where «*type*» is the instance. This convention for naming is adopted so that when an argument `impl` for the implementation of «*type*»'s instantiation of «*class-name*», the term «*class-name*»-«*type*» may be implicitly passed.

[TODO] mention problem of multiple instantiations of same type-class a type?

## 3.3 Constructing Monads

[TODO] reset this section to be more strictly about monad than type-classes in general, which are addressed above.

### 3.3.1 The Monad Type-class

It turns out that we can model the Monad type-class as a parametrized type, with the type `Monad (M:Type) : Type` of terms that implement the monad requirements for $M$. In other words, a term of type `Monad` $M$ "contains" in some way (i.e. *implements*) terms with the types of `bind`, `map`, `lift` as defined in section **??**.

It turns out that we can model the type-class monad as a parametrized type using the intuition established in the previous section. In section 3.1.2, we extracted that the essential capabilities to being a monad are binding, lifting and mapping. So, in the style of type-classes, we can specify a type-class called `Monad` that is the class of types that implement such capabilities:

```
class Monad (M : Type → Type) : Type
  { lift : (α : Type)   ⇒ α → M α
  ; map  : (α β : Type) ⇒ (α → β) → M α → M β
  ; bind : (α β : Type) ⇒ M α → (α → M β) → M β }.
```

Recall that this defines a type `Monad` that represents the type-class monad and also the generalized terms `map`, `lift` and `bind` that work for any type-instance of `Monad`. The following are convenient notations for the generalized terms `bind` and `sequence` (which is implemented in terms of `bind` in the same way explained in section 3.1.1).

Listing 3.3: Notations for binding.

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

let («term-param»*:«type»*) ← («term»1:«type»1) in («term»2:«type»2)
  ::=
    bind «type»1 («type»* → «type»2) «term»1 («term-param»* ⇒ «term»2)
```

The operator `>>=` is right-associative.

Listing 3.4: Notations for sequencing

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

```
do { «term»_1 ; ⋯ ; «term»_n }   ::=   «term»_1 >> ⋯ >> «term»_n
```

The operator >> is right-associative.

Listing 3.5: Notation for binding within do block

```
do { ⟦ «term» ; ⟧_1 ; «term»_* ← «term»_* ; ⟦ «term» ; ⟧_2 }
  ::=
    do { ⟦ «term» ; ⟧_1 ; let «term»_* ← «term»_* in (do { ⟦ «term» ; ⟧_2 }) }
```

[TODO] give brief example

Finally, in order to use these functions with `mutable σ`, we need to construct a term of type `Monad (mutable σ)`. This term is the `Monad` type-class instantiation for `mutable σ`.

Listing 3.6: Instance of the mutable monad

```
instance (σ : Type) ⇒ Monad (mutable σ)
  { map (α β : Type) (f : α → β) (m : mutable σ α) : mutable σ β
      := (s : σ) ⇒
           let (s', a) := m s in
           (s', f a)
  ; lift (α : Type) (a : α) : mutable σ α
      := (s : σ) ⇒ (s, a)
  ; bind (α β : Type) (m : mutable σ α) (fm : α → mutable σ β)
      : mutable σ β
      := (s : σ) ⇒
           let (s', a) := m s in
           fm a s' }.
```

Observe that the constructions of `map`, `lift`, and `bind` are the same as those of `mutable-map`, `mutable-lift` and `mutable-bind` in section 3.1.1. In this concise way, however, we have instantiated the mutability monad without having to name that bunch of `mutable`-specific terms when we actually exclusively want the `Monad`-general terms.

[TODO] show also the un-notated way

### 3.3.2 Mondad Properties

[TODO] describe technical properties that monad capabilities must respect (derived from category theory)

[TODO] equationl properties of monads

## 3.4 Language ℂ

So far we have seen the mutability effect represented as a monad, which is entirely representable in 𝔸 code. We shall define a new language, ℂ, that adapts the monadic strategy for implementing effects in general (including the previous implementation of mutability).

### 3.4.1 Monadic Internal Effects

Constructed monads can fully define the behavior of internal effects. Recall how in 𝔹, each internal effect had a fully-mathematically specified reduction context. In ℂ, the monad instance for each internal effect will play the role that each 𝔹 reduction context played.

#### 3.4.1.1 Exception

In 𝔹, the exception effect was implemented using a reduction context that indicated whether or not there was an exception. Additionally if there was an exception, then the context contained an associated term (the *throw term*). The following definitions inspire a parallel construction in terms of a type `exceptional` that we will soon instantiate as a monad:

Listing 3.7: Definition of `exceptional`

```
type exceptional (ε α : Type) : Type
  := valid : α → exceptional ε α
   | throw : ε → exceptional ε α.
```

Here, `ε` is the *throw type* — the type of throw terms. And, `α` is the *valid type* — the normal type of the expression should no exceptions be thrown. For a fixed exception type `ε`, the type `exceptional ε` (equivalently written `α ⇒ exceptional ε α`) is a monad, parametrized by the valid type.

Listing 3.8: Instance of the exception monad

```
instance (ε : Type) ⇒ Monad (exceptional ε)
  { map (α β : Type)
      (f : α → β) (m : exceptional ε α)
      : exceptional ε β
      := cases m
          { throw e ⇒ throw e
          ; valid a ⇒ valid (f a) }
  ; lift (α : Type)
      (a : α)
      : exceptional ε α
      := valid a
```

```
  ; bind (α β : Type)
      (m : exceptional ε α) (fm : α → exceptional ε β)
      : exceptional ε β
      := cases m
          { throw e ⇒ throw e
          ; valid a ⇒ fm a } }.
```

**3.4.1.1.1 Catching exceptions.** If there was an exception, then the exception was propagated outward to the inner-most `catch` (or to the top level if there was not such a `catch`). Otherwise, reduction was carried out normally.

> [**TODO**] we can actually construct catching here rather than make it primitive and offload to the reduction rules

Listing 3.9: Definition of catching for exceptions

```
term catching (ε α : Type) (k : ε → α) (m : exceptional ε α) : α
  := cases m
      { throw e ⇒ k e
      ; valid a ⇒ a }.
```

Listing 3.10: Notation of catching for exceptions

```
catch { («term»₁:«type»₁) ⇒ («term»₂:«type»₂) } in «term»₃

::=

catching «type»₁ «type»₂ ((«term»₁:«type»₁) ⇒ («term»₂:«type»₂)) «term»₃
```

Notice that we don't need to specify anything analogous to «*exception-name*» from 𝕀𝔹's implementation of the exception effect. This is because correctly matching throws and catches is managed explicitly by the types of the terms involved. A term of type `exceptional int boolean` cannot be managed by a `catch` for `exceptional string boolean` — it wouldn't even type-check! Nested exceptions must be explicitly written as such. For example, an expression that two `string`-exceptions and result type $\alpha$ would have the type `exceptional string (exceptional string α)`.

Revisiting the division-by-0 example, the code should look very similar to 𝕀𝔹, except for the more verbose type signature for `division`.

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

In addition to `throw i` needing to have type `exceptional integer integer`, the other branch of the `if` needs to have the matching type and thus `(i/j)` must be lifted into having type `exceptional integer integer`. The `lift` capability of monads in general achieves this, since `(i,j)` is indeed what we want to result in, and this turns out to be equivalent to `valid` as per the instantiation of the exception monad in listing 3.8.

#### 3.4.1.2 Nondeterminism

[TODO] we *could* implement the nondeterminism effect in the same way as we just did I/O, to parallel how 𝔹 does it. But instead, lets try another approach to nondeterminism: collecting all possible results. This way actually yields nondeterminism as an internal effect.

[TODO] note: this is also the monad instance for `list`

Listing 3.11: The nondeterminisic monad

```
type nondeterministic : Type → Type := list.

instance Monad nondeterministic
  { map f m   := cases m
                   { [] ⇒ []
                   ; a :: m' ⇒ f a :: map f m' }
  ; lift a    := [a]
  ; bind m fm := concat (map fm m) }.
```

[TODO] introduce example of coin-flipping, for reference next chapter

Listing 3.12: Nondeterminism experiment

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

term experiment : nondeterministic boolean
  := let a ← coin-flip in
```

```
  let b ← coin-flip in
  lift (a ∧ b).
```

[TODO] introduce experiment to demonstrate reduction

```
experiment
 ↠
let a ← coin-flip in let b ← coin-flip in lift (a ∧ b)
 ↠
bind coin-flip (a ⇒ bind coin-flip (b ⇒ lift (a ∧ b)))
 ↠
bind coin-flip (a ⇒ bind coin-flip (b ⇒ [a ∧ b]))
 ↠
bind coin-flip (a ⇒ concat (map (b ⇒ [a ∧ b]) [true, false]))
 ↠
bind coin-flip (a ⇒ [[a ∧ true], [a ∧ false]])
 ↠
concat (map (a ⇒ [[a ∧ true], [a ∧ false]]) [true, false])
 ↠
[true ∧ true, true ∧ false, false ∧ true, false ∧ false]
 ↠
[true, false, false, false]
```

### 3.4.2 Monadic External Effects

[TODO] for external effects, things outside the language still need to be appealed to so we can't get away with modelling everything within the language like for the internal effects.

#### 3.4.2.1 I/O

[TODO] The implementation inherits a lot from the 𝔹 implementation: the context $\mathcal{IO}$ and similar reduction rules.

[TODO] remember that Δ is inherited from 𝔹

[TODO] the structure ⟦...⟧ wraps the internal value as io-affected in a way not interactable directly by ℂ code.

Listing 3.13: Instance of the IO monad

```
primitive type io : Type → Type.

primitive term input  (α : Type) : IO α.
primitive term output (α : Type) : α → IO unit.

primitive term io-map  (α β : Type) : (α → β) → io α → io β.
primitive term io-lift (α : Type)   : α → io α.
primitive term io-bind (α β : Type) : io α → (α → io β) → io β.
instance Monad io { map = io-map; lift = io-lift; bind = io-bind }.
```

Π 3.1: Reduction in $\mathbb{C}$: I/O

$$\textsc{Simplify}\quad \frac{\Delta \parallel a \twoheadrightarrow \Delta' \parallel a'}{\Delta \parallel \{\!\{a\}\!\} \twoheadrightarrow \Delta' \parallel \{\!\{a'\}\!\}}$$

$$\textsc{I/O-Map}\quad \texttt{io-map}\ f\ \{\!\{a\}\!\} \twoheadrightarrow \{\!\{f\ a\}\!\}$$

$$\textsc{I/O-Lift}\quad \texttt{io-lift}\ a \twoheadrightarrow \{\!\{a\}\!\}$$

$$\textsc{I/O-Bind}\quad \texttt{io-bind}\ \{\!\{a\}\!\}\ fm \twoheadrightarrow \{\!\{fm\ a\}\!\}$$

$$\textsc{I/O-Join}\quad \{\!\{\{\!\{a\}\!\}\}\!\} \twoheadrightarrow \{\!\{a\}\!\}$$

$$\textsc{Input}\quad \frac{\text{value } v}{\mathcal{IO} \parallel \texttt{input}\ v \twoheadrightarrow \mathcal{IO} \parallel \{\!\{\mathcal{IO}(\texttt{input}\ v)\}\!\}}$$

$$\textsc{Output}\quad \frac{\text{value } v}{\mathcal{IO} \parallel \texttt{output}\ v \twoheadrightarrow \mathcal{IO} \parallel \{\!\{\mathcal{IO}(\texttt{output}\ v)\}\!\}}$$

[**TODO**] notice that, even though I/O is an internal effect, the monadic structure gives it a wrapper that must be managed by the programmer.

[**TODO**] give example of how to organize an I/O program so that its as simple as possible, only needing I/O in particular places at the highest-level and not at each step. Refer to inspiration from last chapter about segregating pure and impure code.

```
main : io unit
  := ...
```

```
main : io unit
  := ...
```

## 3.5 Considerations for Monadic Effects

[TODO] Justify how monads are a good model for thinking about computation being pushed into an implicit context (cite Notions by Moggi).

[TODO] brief explanation of some implementations of monadic effects in the likes of Haskell.

[TODO] But there are issues as demonstrated by The Awkward Squad. Additionally explain problem of composable effects, and how monads have trouble with this

[TODO] End on problem of composing monads.

# K 4

# Algebraic Effect Handlers

## 4.1 Introduction to Algebraic Effect Handlers

In chapter 2, we considered language 𝔹 which implemented effects by introducing specific language features for each kind of effect. In chapter **??**, we considered language 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 𝔸 that provides composable effects while also maintaining the useful features of ℂ.

> [**TODO**] detail the maintained features from 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 𝔹: effects are performed by using specific primitive values, and are handled during reduction to affect the program state. In 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` $\alpha$ for fixed $\alpha$).

## 4.2 Language 𝕀𝔻

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

### 4.2.1 Syntax for 𝕀𝔻

[TODO] english

Π 4.1: Syntax for 𝕀𝔻

| metavariable | constructor | name |
|---|---|---|
| «*declaration*» | `resource` «*resource-name*» ⟦ `(`«*type-param*»`:`«*kind*»`)` ⟧<br>`{` ⟦ «*action-name*» ⟦ `(`«*type-param*»`:`«*kind*»`)` ⟧<br>`:` «*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*»`)` ⟧<br>`:` «*kind*»<br>`{` ⟦ `#`«*action-name*» «*term-parap*» «*term-param*»<br>`⇒` «*term*» `;` ⟧ `}` | handler |
| | «*term*» `with` «*term*» | handle performance |

### 4.2.2 Primitives for 𝕀𝔻

[TODO] english

#### 4.2.2.1 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 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 " ↗ " operator looks similar to the arrow type's arrow, but is tilted upward in appeal to another context.

#### 4.2.2.2 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

$$\texttt{resource}\ \rho\ \{\ g_1\ :\ \alpha_1 \nearrow \beta_1\ ;\ \cdots\ ;\ g_n\ :\ \alpha_n \nearrow \beta_n\ \}.$$

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

Π 4.2: Typing in 𝔻: Resource

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

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`.

#### 4.2.2.3 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:

Π 4.3: Typing in ID: Channel

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

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.

#### 4.2.2.4 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:

Π 4.4: Typing in ID: Performance

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

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 ID.

```
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.

#### 4.2.2.5 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 4.2.2.6) 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 $g : \chi \nearrow \upsilon$ has the form $\#g\ x\ k \Rightarrow b$, 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 $\chi$ 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, $a \Rightarrow a$). Such a value clause that handles relatively-pure values has the form $a \Rightarrow b$, 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 $b \Rightarrow c$, 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 $\alpha$ and has result type $\beta$ is $\alpha \searrow \beta$. The " $\searrow$ " 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
  { #g1 x1 k1 ⇒ b1 ; ⋯ ; gn xn kn ⇒ bn
  ; value av ⇒ bv
  ; final bf ⇒ cf }
```

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

Π 4.5: Typing in 𝕀𝔻: Handler

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

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 $\alpha$ in which `Random`-performances appear, and has result type $\alpha$. The

#### 4.2.2.6 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 4.2.3.

The typing rule is as follows:

Π 4.6: Typing in 𝔻: Handling

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

For example, consider the following term (section 4.2.2.7 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 : α ↘ α` abstracted over all `α : Type`.

**4.2.2.6.1 Nested handlings.** The `with` construction is infixed and is left-associative i.e. `a with b with c` associated to `((a with b) with c)`. For the sake of conciseness, if a term is handled by multiple nested handlers, the entire expression can be abbreviated by the following notation.

Listing 4.1: Notation for nested handlings

```
«term»* with «term»1 with ⋯ with «term»n
  ::=
    «term»* with «term»1, ..., «term»n
```

### 4.2.2.7 Sequencing

Listing 4.2: Construction for sequencing

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

Listing 4.3: 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 𝕀𝔹's for binding within `do` blocks:

Listing 4.4: 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 }) }
```

### 4.2.3 Reduction Rules for 𝕀𝔻

**4.2.3.0.1 Reduction contexts.** The total reduction context $\Delta$ is made up of two subcontexts: $\mathcal{H}$ 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 $\mathcal{P} = [a_3, a_2, a_1]$.

**4.2.3.0.2 Relative Values.** Recall from section 1.2.6 that a term is a value if no reduction rules can simplify it. Note though that terms in 𝕀𝔹 must rely on relatively implicit contexts for reduction — in particular, $\mathcal{H}$. For example: `random1#gen-boolean` $\bullet$ 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` $\bullet$ `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.

**4.2.3.0.3 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.

**4.2.3.0.4 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 $h$.

**4.2.3.0.5 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 `#`$g$ $x$ $k$ `⇒` $b$ 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 $(x\ k \Rightarrow b)\ v\ (a \Rightarrow w)$.

**4.2.3.0.6 Handling value.** The reduction rule HANDLE-VALUE dictates how to use a given handler $h$ to handle a (relative to $h$) value $v$. Let $\texttt{value}\ a \Rightarrow b$ 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 $(a \Rightarrow b)\ v$. 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.

**4.2.3.0.7 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 $\texttt{final}\ b \Rightarrow c$ be the final clause of $h$. The result of HANDLE-FINAL should simply be $c$ with $v$ passed as $b$, written as the application $(b \Rightarrow c)\ v$ (very similar to HANDLE-VALUE).

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

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

Π 4.7: Reduction in 𝕀𝔻

**Simplify**

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

**Simplify**

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

**Sequence**

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

**Bind**

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

**Raise**

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

**Handle**

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

**Perform**

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

**Handle-Final**

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

**Handle-Value**

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

**Handle-Performance**

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

## 4.3 Examples

### 4.3.1 Example: Mutability

[TODO] intro

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

Listing 4.5: Resource for mutability.

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

**4.3.1.0.2 Handler.** 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.

Listing 4.6: Handler for mutability.

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

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.

#### 4.3.1.1 Experiment

In using the mutability effect, the mutability channels correspond to particular mutable memory stores that can be read from via `get` and wrote to via `set`. Say we have a channel `counter : Mutable integer` and we want to construct a term `increment` that adds `1` the counter and results in •. We can write the following program:

```
channel counter : Mutable integer.

term increment : unit → unit
  := do
       { i ← counter#get •
       ; counter#set (i + 1) }.
```

A simple application of `increment` is handled (where $s_0$ is an arbitrary integer), from performance to handled-performance, as follows:

Listing 4.7: Handling a simple `increment` with `initialize`

```
h := initialize s₀ counter.

[h] ; [] ‖ increment •
(Definition)
[h] ; [] ‖ do{ i ← counter#get • ; counter#set (i + 1) }
(Simplify)
[h] ; [] ‖ counter#set ((counter#get •) + 1)
(Perform x2)
[h] ; [(counter#set (@1 + 1), @2), (counter#get •, @1)] ‖ @2
(Handle-Value)
[h] ; [(counter#set (@1 + 1), @2), (counter#get •, @1)] ‖ s ⇒ (s, @2)
(Handle-Performance x2)
[h] ; [] ‖ s ⇒ (s + 1, •)
```

Abbreviate this reduction done by Handle-Performance x2 by Handle-Increment. Now, consider the following experiment using `increment` to be handled.

Listing 4.8: Handling a mutability experiment with `initialize`.

```
h := initialize 1 counter.

[] ; []
  ‖ do{ increment • ; increment • ; counter#get • } with h
[h] ; []
  ‖ do{ increment • ; increment • ; counter#get • }
(Perform x3)
[h] ; [(counter#get •, @3), (increment •, @2), (increment •, @1)]
  ‖ do{ @1 ; @2 ; @3 }
(Handle-Value)
[unvalued(h)] ; [(counter#get •, @3), (increment •, @2), (increment •, @1)]
  ‖ s ⇒ (s, do{ @1 ; @2 ; @3 })
(Handle-Performance)
[unvalued(h)] ; [(increment •, @2), (increment •, @1)]
  ‖ s ⇒ (s, do{ @1 ; @2 ; s })
```

```
(Handle-Increment x2)
[unvalued(h)] ; [(increment •, @2), (increment •, @1)]
  ‖ s ⇒ (s + 2, do{ • ; • ; s + 2 })
(Handle-Final)
[unvalued(h)] ; [] ‖ (3, do{ • ; • ; 3 })
(Simplify)
[unvalued(h)] ; [] ‖ (3, 3)
```

### 4.3.2 Example: Exception

#### 4.3.2.0.1 Resource.

[TODO] description

Listing 4.9: Resource for exception

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

Listing 4.10: Channels for exception

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

#### 4.3.2.1 Experiment

[TODO] think of some experiment

Listing 4.11: 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 }.
```

#### 4.3.2.2 Handling exceptions as optionals

Listing 4.12: 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 4.13: 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
```

### 4.3.3 Example: Nondeterminism

#### 4.3.3.0.1 Resource and channel.

[TODO] describe

Listing 4.14: 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.
```

#### 4.3.3.1 Experiment

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

Listing 4.15: Experiment with nondeterministic coin-flipping.

```
term experiment : boolean
  := (coin#flip •) ∧ (coin#flip •)
```

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

Listing 4.16: 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 4.17: 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]
```

**4.3.3.1.2 Handling rigged flips.** A handler that computes result of experiment where each flip yields heads.

Listing 4.18: Handler for just heads.

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

Listing 4.19: 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
```

**4.3.3.1.3 Handling alternating possibilities.** A more sophisticated handler.

Listing 4.20: 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 4.21: 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
```

### 4.3.4 Example: I/O

Listing 4.22: Resource for I/O

#### 4.3.4.0.1 Resource mad channel.

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

Listing 4.23: Channel for I/O

```
channel io : IO.
```

#### 4.3.4.1 Experiment

[TODO] description

Listing 4.24: Experiment for I/O

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

##### 4.3.4.1.1 Primitive Handler for I/O

[TODO] do I allow this handler as typed? since I don't want to let IO be performed anywhere like this right?

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

Listing 4.25: Handler for standard I/O

```
primitive term standard-io (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 𝒹𝒪 responds to queries with the appropriate responses, matching the specification of the `io` resource. In this way the body of `standard-io` can be imagined as the following:

Listing 4.26: Imaginary body for `standard-io`

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

[TODO] recap specification for 𝒹𝒪

Listing 4.27: Handle greetings with standard I/O

```
h := standard-io io.

[] ; [] ‖ greetings with h
(Simplify)
[] ; []
  ‖ do { name ← io#input •
       ; io#output ("Hello, " ++ name) } with h
(Handle)
[h] ; []
  ‖ do { name ← io#input •
       ; io#output ("Hello, " ++ name) }
(Perform)
[h] ; [(io#input •, a1)]
‖ do { name ← a1
     ; io#output ("Hello, " ++ name) }
(Simplify)
[h] ; [(io#input •, a1)] ‖ io#output ("Hello, " ++ a1)
(Perform)
[h] ; [(io#output ("Hello, " ++ a1),, a2)
    ▷ (io#input •, a1)]
  ‖ a2
(Handle-Value)
[unvalued(h)] ; [(io#output ("Hello, " ++ a1),, a2)
            ▷ (io#input •, a1)]
  ‖ a2
(Handle-Performance)
[unvalued(h)] ; [(io#output ("Hello, " ++ a1),, a2)]
  ‖ (x k ⇒ k 𝒪1(input x)) • (a1 ⇒ a2)
(Simplify)
[unvalued(h)] ; [(io#output ("Hello, " ++ a1),, a2)]
  ‖ (a1 ⇒ a2) 𝒪1(input •)
(Handle-Performance)
[unvalued(h)] ; []
  ‖ (x k ⇒ k 𝒪2(output x)) ("Hello, " ++ a1) (a2 ⇒ (a1 ⇒ a2) 𝒪1(input •))
(Simplify)
[unvalued(h)] ; [] ‖ (a2 ⇒ (a1 ⇒ a2) 𝒪1(input •)) 𝒪2(output ("Hello, " ++ a1
   ))
(Simplify)
[unvalued(h)] ; [] ‖ (a1 ⇒ 𝒪2(output ("Hello, " ++ a1))) 𝒪1(input •)
(Simplify)
[unvalued(h)] ; [] ‖ 𝒪2(output ("Hello, " ++ 𝒪1(input •)))
```

```
(Handle-Final)
[] ; [] ‖ 𝒪₂(output ("Hello, " ++ 𝒪₁(input •)))
(Simplify)
[] ; [] ‖ •
```

Note that the usage of the interface I/O interface $\mathcal{O}$ is subscripted by the order of use. This is necessary since each appearance of $\mathcal{O}_i(\dots)$ only indicates the *new* usage of $\mathcal{O}$ if it is the first appearance of $\mathcal{O}_i$. Otherwise, it merely references the result of the first usage of $\mathcal{O}_i$.

[**TODO**] note how last reduction to unit-term is given by specification of $\mathcal{O}$

**4.3.4.1.2 Contructed Handler for I/O** The previous section made use of a primitive, implicit handler for the I/O which appealed to a language-external interface 𝒽𝒪. However, one of the freedoms granted by algebraic effect handlers is the ability to implement a variety of handlers for the same effect. So, we can handle the same experiment `greetings` using a constructed, explicit handler. Call such a handler a *pure* handler for I/O.

The I/O effect requires some stateful stream of inputs, and a store for outputs. We can represent this stream and store each as a list — the I/O handler is parametrized by a list of inputs and results in the computation's value and a list of outputs. The following handler implements this specification:

Listing 4.28: Handler for pure I/O

```
term pure-io (α:Type) (inputs : list string) : IO → α ↘ (list string × α)
  := handler
       { #output x k ⇒ (inputs outputs ⇒ k • inputs (outputs ◇ [x]))
       ; #input  _ k ⇒ (inputs outputs ⇒
                          k (head inputs) (tail inputs) outputs)
       ; value     a ⇒ (inputs outputs ⇒ (a, outputs))
       ; final     f ⇒ f inputs [] }.
```

[**TODO**] describe how it works similarly to the mutability handler initialize

[**TODO**] note that if there arent enough elements in inputs, then head inputs will cause run-time error

## 4.4 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

Σ 4.1: Reduction Test

| | | |
|---|---|---|
| | `[] ; []` | `greetings with h` |
| (S.) | | |
| | `[] ; []` | `io#output ("Hello, "++(io#input •)) with h` |
| (H.) | | |
| | `[h] ; []` | `io#output ("Hello, "++(io#input •))` |
| (P.) | | |
| | `[h] ; [(io#input •, a1)]` | `io#output ("Hello, "++a1)` |
| (P.) | | |
| | `[h]`<br>`; [(io#output ("Hello, " ++ a1) , a2)`<br>`  ,(io#input •, a1)]` | `a2` |
| (H.V.) | | |
| | `[unvalued(h)]`<br>`; [(io#output ("Hello, " ++ a1), a2)`<br>`  ,(io#input •, a1)]` | `is os ⇒ (os, a2)` |
| (H.P.) | | |
| | `[unvalued(h)] ; [(io#input •, a1)]` | `(x k is os k • is (os ◇ [x]))`<br>`  ("Hello, "++a1) (a2 is os ⇒ (os, a2))` |
| (S.) | | |
| | `[unvalued(h)] ; [(io#input •, a1)]` | `is os ⇒ (os ◇ ["Hello, "++a1], •)` |
| (H.P.) | | |
| | `[unvalued(h)] ; []` | `(_ k is os ⇒ k (head is) (tail is) os)`<br>`  • (a1 is os ⇒ (os ◇ ["Hello, "++a1], •))` |
| (S.) | | |
| | `[unvalued(h)] ; []` | `is os ⇒ (os ◇ ["Hello, "++"head is], •)` |
| (H.F.) | | |
| | `[] ; []` | `(is os ⇒ (os ◇ ["Hello, "++"head is], •))`<br>`  ["Henry", "Blanchette"] []` |
| (S.) | | |
| | `[] ; []` | `(["Hello, Henry"], •)` |

# K 5

# Freer-Monadic Effects

## 5.1 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 ℂ. For 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 4, 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 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 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 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.

## 5.2 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 5.1: 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 free*r* monads do not require `υ` to be a functor, they yield a monad instance "for free*r*."

```
  := 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 $\alpha$ to the impure type `Freer υ α`[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 $\upsilon$ 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 $(\chi \to$ `Freer` $\upsilon\ \alpha)$-continuation, in order to produce a `Freer υ α`. The intuition is that `impure y k` encodes a $\upsilon$-wrapped $\chi$-term and a continuation $k$ that is waiting for an (unwrapped) $\chi$-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 υ α`). Such a handler must depend on the structure of the base type $\upsilon$, and so much be implemented separately for each $\upsilon$. A **freer-monadic handler**, or just **handler**, has a type of the form `Freer υ α → ω α`, where $\omega$ is the type of affected results (parametrized by the result type $\alpha$). 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 5.2: 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 $\mathbb{C}$) 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 $\upsilon$.

So far we have defined the `Freer` type as a way of lifting a type $\upsilon$ `: Type → Type` to a monad instance `Freer` $\upsilon$ (parametrized by result type $\alpha$). 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.
```

## 5.3 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** $\upsilon$. We shall call a term of type `Freer` $\upsilon$ $\alpha$ a freer-$\upsilon$-computation with $\alpha$-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 5.3: 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 → σ × α` function that acts as a handler of the mutability effect.

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

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

[TODO] mutability example from prev chapters

```
type store := integer × integer × boolean.

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

initialize (1, 2, true)
  experiment
↠ (Definition of experiment)
initialize (1, 2. true)
  (get • >>= (s ⇒
    let x := part-1 s in
    let y := part-2 s in
    set (x, y, x < y)))
↠ (Definition of get, set)
initialize (1, 2, true)
  (impure (s ⇒ (s, s)) pure >>= (s ⇒
    let x := part-1 s in
    let y := part-2 s in
    impure (_ ⇒ ((x, y, x < y), •)) pure))
↠ (Apply bind)
initialize (1, 2, true)
  (impure (s ⇒ (s, s)) (s ⇒
    pure s >>= (s ⇒
      let x := part-1 s in
      let y := part-2 s in
      impure (_ ⇒ ((x, y, x < y), •)) pure)))
↠ (Apply bind)
initialize (1, 2, true)
  (impure (s ⇒ (s, s)) (s ⇒
    let x := part-1 s in
    let y := part-2 s in
    impure (_ ⇒ ((x, y, x < y), •)) pure))
↠ (Apply initialize)
let (s', _) := (s ⇒ (s, s)) (1, 2, true) in
```

```
initialize s'
  (let x := part-1 (1, 2, true) in
   let y := part-2 (1, 2, true) in
   impure (_ ⇒ ((x, y, x < y), •)) pure)
↠ (Simplify)
initialize (1, 2, true)
  (impure (_ ⇒ ((1, 2, false), •)) pure)
↠ (Apply initialize)
let (s', a) := (_ ⇒ ((1, 2, false), •)) (1, 2, true) in
initialize s' (pure a)
↠ (Simplify)
initialize (1, 2, false) (pure •)
↠ (Simplify)
((1, 2, false), •)
```

## 5.4 Examples of Freer-Monadic Effects

### 5.4.1 Example: Exception

The exception effect has base type $\varepsilon \oplus \alpha$, where $\varepsilon$ is the exception type and $\alpha$ 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 : ε → α`, 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 5.4: 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 5.5: 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
```

### 5.4.2 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 4. 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`.

```
all-possibilities experiment
↠ (Definition of 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]
```

### 5.4.3 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 }.
```

**5.4.3.0.1 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.
```

**5.4.3.0.2 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 y k ⇒ let (is', os, a) := y 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 (ιo : Type → Type) {IO ιo} : io ιo unit
  := do
      { name ← input •
      ; output ("Hello, " ++ name) }.

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

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

## 5.5 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

## 5.6 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

# Π A

# Appendix: Prelude for 𝔸

## A .1 Functions

```
term identity (α:Type) (a:α) : α : a.

term compose (α β γ : Type)
  : (α → β) → (β → γ) → (α → γ)
  := f g a ⇒ f (g a).
```

## A .2 Data

Listing A .1: unit

```
// unary sum
type unit : Type := • |.
```

Listing A .2: boolean

```
type boolean : Type := true | false.
```

Listing A .3: natural number

```
type natural : Type
  := zero : natural
```

```
  | succ : natural → natural.

// canonical terms
term N0 : natural := zero.
term N1 : natural := succ N0.
term N2 : natural := succ N1.
term N3 : natural := succ N2.
// ⋯ and so on

// addition
term add (m n : natural) : natural
  := cases m
       { zero    ⇒ n
       ; succ m' ⇒ succ (add m' n) }.

// multiplication
term mul (m n : natural) : natural
  := cases m
       { zero    ⇒ zero
       ; succ m' ⇒ add n (mul m' n) }.

// minimum
term min (m n : natural) : natural
  := cases m
       { zero    ⇒ zero
       ; succ m' ⇒ cases n
                     { zero    ⇒ m'
                     ; succ n' ⇒ succ (min m' n') } }.

// maximum
term max (m n : natural) : natural
  := cases m
       { zero    ⇒ n
       ; succ m' ⇒ cases n
                     { zero    ⇒ m
                     ; succ n' ⇒ succ (max m' n') } }.
```

Listing A .4: integer

```
type integer : Type
  := negative : natural → integer
```

```
  | positive : natural → integer.

// canonical terms
basic term  0 : integer := positive N0.
basic term  1 : integer := positive N1
basic term -1 : integer := negative N1
basic term  1 : integer := positive N2
basic term -1 : integer := negative N2
// ⋯ and so on

// addition
term add (i j : integer) : integer
  := cases i
       { positive m ⇒ cases j
                      { positive n ⇒ add m n
                      ; negative n ⇒
                      }
       }

// TODO: addition
term add (i j : integer) : integer
  := cases i
       { left m ⇒ cases j
                  { left  n ⇒ if max m n ==
                  ; right n ⇒  } }

// negate
term neg (i : integer) : integer
  := cases i
       { left  n ⇒ right n
       ; right n ⇒ left  n }.

// subtraction
term sub (i j : integer) : integer := add i (neg j).

// modulus
term mod (i j : integer) : integer
  := if i < j
       then i
       els mod (i - j) j.
```

Listing A .5: rational

```
type rational : Type := natural × natural.

basic term 0   : rational := (0, 1).
basic term 1   : rational := (1, 1).
basic term 1/2 : rational := (1, 2).
basic term 2/8 : rational := (2, 8).

// TODO: division
term div (q r : rational) : rational
  := ⋯
```

Listing A .6: Notations for infixed binary numerical operations.

```
«term»₁ + «term»₂   ::=   add «term»₁ «term»₂

«term»₁ * «term»₂   ::=   mul «term»₁ «term»₂

- «term»   ::=   neg «term»

«term»₁ - «term»₂   ::=   sub «term»₁ «term»₂

«term»₁ / «term»₂   ::=   div «term»₁ «term»₂
```

**A .2.0.0.1 Characters and Strings.** These data types are introduced primitively because it is much to annoying and irrelevant to formally introduce them here.

Listing A .7: string

```
primitive type character.
primitive type string.

// some utilities
primitive term character-join : list character → string.
primitive term string-join : string → string → string.
primitive term string-concat : list string → string.
primitive term string-explode : string → list character.
primitive term character-append : character → string → string.
primitive term character-unappend : string → character × string.
\end{string}
```

```
\begin{notational}[caption={Notations for characters and strings.}]
«term»_1 ++ «term»_2   ::=   string-concat [«term»_1, «term»_2]
\end{notational}

%
    ----------------------------------------------------------------------------

%
    ----------------------------------------------------------------------------

%
    ----------------------------------------------------------------------------

\section{Data Structures}

\begin{program}[caption={optional}]
type optional (α : Type) : Type := α ⊕ unit.

basic term some (α : Type) (a : α) : optional α := left a.
basic term none (α : Type) : optional α := right •.
```

Listing A .8: list

```
type list (α : Type) : Type := unit ⊕ (α × list α).

term nil (α : Type) : list := left •.
term app (α : Type) (a : α) (as : list α) := right (a, as).
```

Listing A .9: list notations

```
[]   ::=   nil

«term»_1 :: «term»_2   ::=   app «term»_1 «term»_2

«term»_1 ◇ «term»_2   ::=   concat «term»_1 «term»_2
```

Listing A .10: list utilities

```
term concat (α : Type) (ass : list (list α)) : list α
  := cases ass
       { []        ⇒ []
       ; as :: ass' ⇒ cases as
                        { []      ⇒ concat ass'
                        ; a :: as' ⇒ a :: concat (as' :: ass') }.

// undefined on nil
term head (α : Type) (as : list α) : α
  := cases as{ a :: _ ⇒ a }.

term tail (α : Type) (as : list α) : list α
  := cases as{ _ :: as' ⇒ as' }.

term contains (α : Type) (as : list α) (a : α) : boolean
  := cases as
    { []       ⇒ false
    ; a' :: as' ⇒ if a = a' then true else contains as' a }.

term repeat (α : Type) (n : natural) (a : α) : list α
  := cases n
       { zero => []
       ; succ n' => a :: repeat n' a }.

% never use functors by themselves anymore
% instance Functor list :=
%   { map a f as
%       := cases as
%            { []      ⇒ []
%            ; a :: as' ⇒ f a :: map f as' } }.

instance Monad list :=
  { lift a := [a]
  ; map f as := cases as
                  { []      ⇒ []
                  | a :: as' ⇒ f a :: map f as' } }.
  ; bind m fm := concat (map fm m) }
```

Listing A .11: tree and forest

```
type tree (α : Type) : Type := α × list (tree α).
```

```
type forest (α : Type) : Type := list (tree α).
```

Listing A .12: mapping

```
type mapping (α β : Type) : Type := α → β.

term make-mapping (α β : Type) (ls : list (α × β)) : mapping α β
  := ⋯

term lookup (α β : Type) (m : mapping α β) (a : α) : β
  := m a.
\end{mapping}

\begin{notational}[caption={mapping notations}]
«type»_1 ⇻ «type»_2    ::=    mapping «type»_1 «type»_2.

⟦ «term»_i1 ↦ «term»_i2 ⟧    ::=    make-mapping [⟦ («term»_i1, «term»_i2) ⟧]

«term»_1@«term»_2    ::=    lookup «term»_1 «term»_2.
\end{notational}

%
    ------------------------------------------------------------------------------

\section{Classes}

\begin{program}[caption={Equalible}]
class Equalible (α : Type) : Type
  { (==) : α → α → boolean }.
```

Listing A .13: Equalible instances

```
instance Equalible unit := {⋯}.
instance Equalible boolean := {⋯}.
instance Equalible natural := {⋯}.
instance Equalible integer := {⋯}.
instance Equalible rational := {⋯}.
instance (α β : Type) {Equalible β} {Equalible α} ⇒ Equalible (α ⊕ β) := {
    ⋯}.
```

```
instance (α β : Type) {Equalible β} {Equalible α} ⇒ Equalible (α × β) := {
   ⋯}.
instance (α:Type) {Equalible α} ⇒ Equalible (optional α) := {⋯}.
instance (α:Type) {Equalible α} ⇒ Equalible (list α) := {⋯}.
instance (α:Type) {Equalible α} ⇒ Equalible (tree α) := {⋯}.
instance (α:Type) {Equalible α} ⇒ Equalible (forest α) := {⋯}.
instance (α:Type) {Equalible α} ⇒ Equalible (mapping α) := {⋯}.
```

# Π B

# Appendix: Prelude for 𝔹

# Π Γ

# Appendix: Prelude for ℂ

```
term join : Monad M => M (M α) -> M α
  := mm ⇒ mm >>= identity
```

# Π Δ

# Appendix: Prelude for 𝕀𝔻

# Bibliography

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