Chapter 1

Monadic Effects

1.1 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

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. 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.

1.1.1 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, mutable table of variables. This implementation required, however, several new syntactical structures and reduction rules. Is there a way to implement something similar without adding so many extensions of 𝔸?

> [**TODO**] introduce terminology: σ-stateful α-computation.

[TODO] decide and refactor for terminology: what exactly 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 σ-stateful α-computation is a stateful computation where the state is of type σ and the result is of type α. 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 σ-state to a pair of the affected σ-state and the α-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 σ-computation that does not modify the state and lifts the state, `set` is a σ-computation that replaces the state with a given $s' : \sigma$ and lifts $\bullet$. In this way, using `get` and `set` in 𝔸 fills exactly the same role as a simple 𝔹 mutable effect of one mutable variable.

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

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 σ-computations m `: mutable σ α` and m' `: mutable σ β` the sequenced σ-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'.
```

Binding. However, in this form it becomes clear that `mutable-sequence` throws away some information — the α-result of the first stateful computation. To avoid this amounts to allowing m' to reference m's result, and can be modeled by typing m' instead as $\alpha \to$ `mutable` $\sigma\ \beta$. A sequence that allows this is called a *binding-sequence*, or just **bind**, since it sequences m, m' and also *binds* the first parameter of m' to the result of m.

```
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'.

So far, we have defined all of the *special* operations that 𝔹 provides for stateful computations. 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" σ-stateful computations of α using the type `mutable` $\sigma\ \alpha$ rather than just an "unwrapped" type α. What is still missing in our implementation is any way to trivial *embed* non-stateful computation within stateful computations. There are two kinds of such embeddings: *lifting* and *mapping*.

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 $\alpha \to$ `mutable` $\sigma\ \alpha$.

```
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
  ≔ let old ← get in
    set 0 >> lift old.
```

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 $\alpha \to \beta$ to a function of type `mutable` $\sigma\ \alpha \to$ `mutable` $\sigma\ \beta$. Call this function `mutable-map`.

> [TODO] Choose which implementation to use. (1) is very slick but hard to understand. (2) is a little more easy to understand but makes use of `get`. (3) is very easy to understand and falls in line with the other `mutable-⋆`.

```
term mutable-map (σ α β : Type) : (α → β) → mutable σ α → mutable σ β
  ≔ get >>= (lift ∘ f).
```

[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 $\mathbb{C}$, Appendix A).

[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*.

```
term mutable-map (σ α β : Type) : (α → β) → mutable σ α → mutable σ β
  ≔ f m ⇒ let a ← get in
           (lift ∘ f) a.

term mutable-map (σ α β : Type) : (α → β) → mutable σ α → mutable σ β
  ≔ f m ⇒
      s ⇒ let (s', a) ≔ m s in (s', f a)
```

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.

1.1.2 Formalizion of Monad

Definition of Functor

In section 1.1.1, we described the mapping capability of a the mutability monad. Though they are not necessarily monads, it turns out that the class of structures with at least the mapping capability are interesting on their own as well. We shall formalize this class here as part of the process of formalizing monad, and also for useful future reference.

> A **functor** is a structure with a mapping capability i.e. functors can be mapped over.

Since monads must also have such a mapping capability, all monads are also functors.

Definition of Monad

In section 1.1.1, 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.

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?

1.2 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.

1.2.1 Concept of Classes

The concept of a *classes* 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 the behavior of an 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 how in order for a type to be a monad, it must provide certain capabilities. 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 void called `eat` and `sleep`.

```
abstract class Animal {

  abstract public void eat();
  abstract public void 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 void eat()   { println(this.name + " eats kibble."); }
  public void sleep() { println(this.name + " naps."); }
```

[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 void hunt()  { println(this.name + " hunts for mice."); }

}

class Dog extends Animal {

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

  public void eat()   { println(this.name + " eats steak."); }
  public void sleep() { println(this.name + " sleeps wrestlessly."); }
  public void walk()  { println(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`.

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.*

1.2.2 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 Animal (α : Type)
  ≔ α → unit → unit // eat
  × α → unit → unit // sleep
```

A term of type `Animal` A for some type A would be 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 projection of `impl` is taken where i is the index of the capability in the product. Likewise instantiating `Cat` and `Dog` for `Animal` is written as the following:

```
type cat
  := string           // name
  × (unit → unit)  // eat
  × (unit → unit). // sleep

// cat is an instance of Animal
term Animal-cat : Animal cat
  := ( 2-nd    // eat
     , 3-rd ). // sleep

// dog is an instance of Animal
term Animal-dog : Animal dog
  := ( 2-nd    // eat
     , 3-rd ). // sleep.
```

Notations for type-classes

[TODO] english transition

Type-class Definition. Defining a type-class requires specifying the terms that must be implemented for each instance of the type-class. The following notation allows concise specification, as well as additionally generating the terms, generalized to all instances of the type-class, that are specified.

Listing 1.1: Notation for type-class definition

```
class «class-name» («type-param»:«Type») : «Type»
  { ⟦ «term-name»_i : «type» ; ⟧ }.

::=

type «class-name» («type-param»:«Type») : «Type» := ⟦ «type»_i ×⟧.
⟦ term «term-name»_i («type-param»:«Type») (impl : «class-name» «type-param»)
    : «type»_i
    := i-th impl. ⟧
```

Type-class Instantiation. Instantiating the a type A 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 1.2: Notation for type-class instantiation

```
instance ⟦(«type-param»_i : «Type»_i) ⇒⟧ «class-name» «type»
  { ⟦ «term-name»_j : «type»_j := «term»_j ;⟧ }.

::=

⟦ term «class-name»-«type»
    : ⟦(«type-param»_i : «Type»_i) ⇒⟧ «class-name» «type»
    := ⟦ «term»_j × ⟧. ⟧
```

[TODO] explain how the way this notation works is a little tricky, since «*type*» may not be in a valid term-name format.

1.2.3 Conventions for using Type-classes

[TODO] checking passively for type-class instances, providing them implicitly when necessary? This is kind of a larger change though.

1.3 Constructing Monads

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

1.3.1 The Functor Type-class

[TODO] english description.

Listing 1.3: Definition of the Functor type-class

```
class Functor (F : Type → Type) : Type
  { map (α β : Type) : (α → β) → (F α → F β) }.
```

1.3.2 The Monad Type-class

[TODO] reword

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 `map`, `lift`, and `bind` as defined in section **??**. The simplest way to represent such an *implementation* of a type-class instance in this way is to represent the implementation as a type-product of the types of the required terms. Concretely, for the `Monad` type-class, we specify the `Monad` type as

[TODO] consider removing this part where I do it without the notation, since I've already defined the notation well in the previous section.

```
type Monad (M : Type → Type) : Type
  := ((α β : Type) ⇒ (α → β) → M α → M β)       // map
  × ((α : Type)   ⇒ (α → M α))                  // lift
  × ((α β : Type) ⇒ (M α → (α → M β) → M β )). // bind
```

Using the notation from listing 1.1, the definition of the `Monad` type-class is written more aesthetically as:

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

This allows working with monads (and other type-classes) using the following intuition: given a term of type `Monad` M, the terms required to be constructed in order for M to be a monad are available by projecting the from `Monad` M as the product of those types. Thus we can define `map`, `lift`, and `bind` for monads in general as follows:

```
term map (M : Type → Type) (α β : Type) (impl : Monad M)
  : (α β : Type) ⇒ (α → β) → M α → M β
  := 1-st impl.

term lift (M : Type → Type) (α : Type) (impl : Monad M)
  : (α : Type) ⇒ α → M α;
  := 2-nd impl.
```

```
term bind (M : Type → Type) (α β : Type) (impl : Monad M)
  : (α β : Type) ⇒ M α → (α → M β) → M β
  ≔ 3-rd impl.
```

[**TODO**] define notation for these, such as » and »= and let ← in

Listing 1.4: Notations for binding.

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

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

Listing 1.5: Notation for sequencing.

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

do { ⟦ «term»ᵢ ;⟧ }    ::=   ⟦ «term»ᵢ >>⟧
```

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 *implementation* for `mutable σ`.

```
term Monad-mutable
  : (σ : Type) ⇒ Monad (mutable σ)
  ≔ ( // map
      (α β : Type) (f : α → β) (m : mutable σ α) ⇒
        (s : σ) ⇒
          let (s', a) ≔ m s in
          (s', f a)
      // lift
    , (α : Type) (a : α) ⇒
        (s : σ) ⇒ (s, a)
      // bind
    , (α β : Type) (m : mutable σ α) (fm : α → mutable σ β) ⇒
        (s : σ) ⇒
          let (s', a) ≔ m s in
          a s' ).
```

This term can be passed as the first term argument to `map`, `lift`, and `bind` to yield terms that are concretely compatible with the `mutable σ` instance of the `Monad` type-class.

Using the notation from listing 1.2, the implementation for the `Monad` instance of `(σ : Type) ⇒ State σ` is written more cleanly as:

Listing 1.6: 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' }.
```

1.4 Definition of 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).

1.4.1 Monadic Internal Effects

[TODO] The internal effects we are using as examples are mutability and exception. Since they are internal, monad itself can supply the structure that yields an implicit contents for working within monadic effects, but translates to a language-explicit monadic handling of the effects.

Exception

Listing 1.7: Instance of the exceptional monad

```
type exceptional (ε α : Type) : Type ≔ ε ⊕ α.

basic term raise (ε α : Type) : ε → exceptional ε α ≔ left.
basic term valid (ε α : Type) : α → exceptional ε α ≔ right.
```

```
instance (ε : Type) ⇒ Monad (exceptional ε)
  { map (α β : Type) (f : α → β) (m : exceptional ε α)
      : exceptional ε β
      ≔ case m
          { raise e ⇒ raise e
          ; valid a ⇒ valid (f a) }
  ; lift (α : Type) (a : α) : exceptional ε α
      ≔ okay a
  ; bind (α β : Type) (m : exceptional ε α) (fm : α → exceptional ε β)
      : exceptional ε β
      ≔ case m
          { raise e ⇒ raise e
          ; valid a ⇒ fm a } }.
```

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 1.8: The nondeterminisic monad

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

// a nondeterministic result from a given selection
term sample : list α → nondeterministic α
  ≔ identity.

instance Monad nondeterministic
  { map f m   ≔ case 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 1.9: Nondeterminism experiment

```
term coin-flip : nondeterministic boolean
  ≔ sample [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]
```

1.4.2 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.

I/O

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

[**TODO**] the structure {{...}} wraps the internal value as io-affected in a way not interactable directly by $\mathbb{C}$ code.

Listing 1.10: 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 }.
```

Table 1.1: Reduction in $\mathbb{C}$: I/O

Simplify	$\dfrac{\Delta \parallel a \twoheadrightarrow \Delta' \parallel a'}{\Delta \parallel \{\!\{a\}\!\} \twoheadrightarrow \Delta' \parallel \{\!\{a'\}\!\}}$
I/O-Map	$\texttt{io-map}\ f\ \{\!\{a\}\!\} \twoheadrightarrow \{\!\{f\ a\}\!\}$
I/O-lift	$\texttt{io-lift}\ a \twoheadrightarrow \{\!\{a\}\!\}$
I/O-Bind	$\texttt{io-bind}\ \{\!\{a\}\!\}\ fm \twoheadrightarrow \{\!\{fm\ a\}\!\}$
I/O-Join	$\{\!\{\{\!\{a\}\!\}\}\!\} \twoheadrightarrow \{\!\{a\}\!\}$
Input	$\dfrac{\text{value } v}{\mathcal{IO} \parallel \texttt{input}\ v \twoheadrightarrow \mathcal{IO} \parallel \{\!\{\mathcal{IO}(\texttt{input}\ v)\}\!\}}$
Output	$\dfrac{\text{value } v}{\mathcal{IO} \parallel \texttt{output}\ v \twoheadrightarrow \mathcal{IO} \parallel \{\!\{\mathcal{IO}(\texttt{output}\ v)\}\!\}}$

[**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
  ≔ ...
```

1.5 Considerations

[**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. But there are issues as demonstrated by The Awkward Squad.

[**TODO**] explain problem of composable effects, and how monads have trouble with this