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

May 4, 2020

Approved for the Division
(Computer Science)

Jim Fix

Table of Contents

Abstract

There exists a dilemma of two programming language design philosophies: *imperative* languages and *declarative* languages. Imperative languages are well-adapted for real-world programming, one reason being their ability to interact with implicit *implicit contexts*, which is formalized by the concept of *performing effects*. Though effects can be very useful, they are often *dangerous* — effects yield complexly-interdependent behaviors that are hard to analyze and predict. Declarative languages are less well-adapted for this imperative style of programming, but can be much more *safe* — behavior is locally contained and easily analyzable and predictable.

This thesis approaches the challenge of designing a declarative programming language that provides effects with all the same capabilities of imperative language's effects while still maintaining safety. The design process follows a progression of five languages: (1) Func is a foundational declarative language, (2) Impe extends Func with imperative effects, (3) Mona extends Func with monadic effects, (4) Alge extends Func with algebraic effects with handlers, and finally (5) Frer extends Mona with a freer-monadic effects (a monadic implementation of algebraic effects with handlers). The effect framework of Frer meets the goals of the design challenge, and a subsequent discussion analyzes its significance.

Chapter 1

Pure Computation

1.1 Introduction

In 1936, Alan Turing proposed *Turing machines*[1] as a formal representation of *computation*. A Turing machine performs computation by taking as input a string of symbols (a *program*), reading the symbols, changing its own internal state via some pre-defined rules prescribed by its construction, and writing an output. A Turing machine's *program* amounts to a series of instructions dictating the order and specifics of these activities (e.g. what to write, what to read, which state to change to). The language style resulting from using this model as a programmer's interface to computation will be detailed in chapter 2 as *imperative programming*.

Also in 1936, Alonzo Church proposed the *λ-calculus*[2] as a formal representation of *computation*. In contrast to Turing machines, the λ-calculus exists purely as a language of formulae with no interactive, stateful machine attached. The λ-calculus represents computation as the process of transforming λ-calculus-terms into other λ-calculus-terms according to *rewrite*, or *reduction*, rules. A λ-calculus *program* is a term that, when provided the proper inputs, reduces to the function's output. In this way, λ-calculus programs look much more like mathematical functions than computer code. The language style resulting from using this model as a programmer's interface to computation will be detailed in chapter 2 as *declarative programming*.

Both of these highly abstract systems are still used prolifically in the theoretical computer science community, but how well do they map on to how real-world programmers treat computers?[3] On the one hand, Turing machines at a high level behave very similarly to programmers' interface to computers, though the lowest-level activities dictated by a real-world computer are more complicatedly expressed.

---

[1] Turing first described Turing machines, but did not name them such, in [13].

[2] The *λ-calculus* was first used in [4]. Importantly for this work however, the *simply-typed λ-calculus* was introduced in 1940 by Church in [5].

[3] I do not mean to claim that the original purpose of the systems was to map onto future programming languages. However, the comparison is useful to see what is commonly used in real-world programming but missing in the theoretical framework.

On the other hand, the λ-calculus is almost completely[4] removed from any reasonable description of real-world programs. Only very small sections, such as simple mathematical operations, could be feasibly translated into λ-calculus terms. Why have real-world programming languages so heavily adopted a Turing-machine-like sequential-instruction model and not a λ-calculus-like formulaic model for representing computation?

Though the simply-typed λ-calculus has been proven to be exactly as *expressive* as Turing machines (that is, there is a correspondence between pairs of Turing machines with their programs and λ-calculus terms), the λ-calculus requires the entire behavior of each term to be specified within the term itself, whereas a Turing machine can make use of its state to keep track of external contexts when reading each individual instruction of a program. Real-world languages for real-world software heavily rely on externally-defined behavior, and so the Turing machine model better fits this need. This observation is the inspiration for the concept of *effects*, which will be detailed in chapter 2.

However, the λ-calculus model still has significant benefits that are desirable yet lacking in many real-world programming languages. λ-calculus is a very **safe** language — that is, its programs are very easy to formally analyze and predict the behavior of. This is in contrast to the **dangerous** inescapable complexity of implicit dependencies on implicit contexts and externally-defined behavior of real-world languages, which is very difficult to formally analyze, test, and predict the behavior of. This safety, though, is easily lost when the features needed for real-world programs — namely, effects — are added to a λ-calculus-like language (as will be demonstrated by Impe). The goal of this thesis is to consider an alternative to this predicament by introducing a general way of implementing such features in a λ-calculus-like language while still preserving safety. The rest of this chapter introduces Func, a slightly-embellished version of the typed λ-calculus that will serve as a basis for the later extensions.

Chapter 2 expands the concept of and programming-language relevance of the concept of *effects* (i.e. the features missing yet desired for real-world programming languages). It then introduces Impe (extension of Func), which takes a simple approach in implementing *imperative* effects, using a selection of example effects: mutability, exception, nondeterminism, and input/output. It ends with some considerations for imperative-effect framework. Pros: it is good for quick and dirty programs that correspond closely to imperative languages' (in fact, the real-world language OCaml uses this effect framework). Cons: it *dangerously* lacks *explicitly-typed* effects, programmable *internal* effects, and a generalized language-structure for defining effects.

Chapter 3 introduces the concept of *monad* (and its corresponding type-class) as a way to represent effects within Func. It then introduces Mona (a minimal extension of Func), which constructs a monad instance for each effect. It ends with some considerations for the monadic-effect framework: Pros: it is very *safe* (esp. *type-safe*; has explicitly-typed effects), promotes programmable internal effects, and does not rely on any reduction contexts. Cons:

[4]Though, importantly, there are a few programming language communities making use of λ-calculus inspired languages such as Haskell, OCaml, and of course the legendary Lisp language.

it requires a lot of complicated type considerations from the programmer and does not allow the *composition* of effects.

Chapter 4 introduces the concept of *algebraic effects with handlers* as an alternative way (to monads) to extend Func with effects while still preserving some safety. It then introduces Alge (major extension of Func), which provides a collection of language structures and reduction rules for hosting algebraic effects and handlers. It ends with some considerations for the algebraic effect and handler framework. Pros: It preserves some of the safety of Func without becoming as type-burdensome as Mona, allows for the composition of effects (at the *performance* level), and allows the definition of multiple ways to *handle* the same effect performances. Cons: it loses some important type-safety managed by Mona and relies heavily on implicit reduction contexts.

Finally, chapter 5 introduces the concept of *freer monads* as an implementation of algebraic effects and handlers into Mona as a monad. It then introduces Frer (minimal extension of Mona), which constructs a single freer monad instance as a general monad for all effects. It demonstrates how Frer can be used, with a *stacked-freer monad*, to allow the composability of freer-monadic effects in a way mimicking algebraic effects and handlers. It ends with some considerations for the freer-monadic effect framework. Pros: it combines the pros of monadic effects and algebraic effects with handlers, while still maintaining the type-safety of Func. Cons: it still requires some complicated type considerations in the background (although they need not be too explicitly exposed to the programmer in order to typically use them).

1.2 Language Func

This section presents the language Func, a slightly-embellished version of the typed λ-calculus. There are three parts to working with Func programs: *syntax*, *typing*, and *reduction*. Syntax defines a grammar for what kinds of expression are **well-formed** i.e. are meaningful strings of symbols. Typing defines an inference scheme for which kinds of well-formed are **well-typed** i.e. strings that correspond to valid programs. Finally, reduction defines the logical steps used for **evaluating** well-typed terms i.e. computing.

Keep in mind that Func and all other languages defined in this thesis are *not* sensitive to the white-space (i.e. spaces and line-returns), which is manipulated freely for the sake of readability. For example, the string

```
a (b c)
  d
  (e
    f)
```

is read as equivalent to `a (b c) d (e f)`. This is not the norm in many programming languages (e.g. Python).

1.2.1 Syntax

Table 7.1 fully defines the syntax Func. The rest of the section explains the significance of and how to interpret Func programs.

Table 1.1: Syntax for Func

metavariable	generator	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*»	kind	atom
	kind → «*kind*»	arrow
«*type*»	«*type-name*»	atom
	(«*type-param*»:«*kind*») ⇒ «*kind*»	function
	«*kind*» «*kind*»	application
«*term*»	«*term-name*»	atom
	(«*term-param*»:«*type*») ⇒ «*term*»	function
	«*term*» «*term*»	application

Declarations are written at the top-level of a program, and declare globally-accessible **names**. Names can be declared either *primitively* or *constructively*. A **primitive** declaration is axiomatic in that it only declares the type of a new term name (or the kind of a new type

name) without defining the term or type that the name is equal to. For example, the type `unit` and its one term can be introduced primitively declared by the following.

```
primitive type unit : kind.

primitive term • : unit.
```

A **constructive** declaration does provide the term or type that a new term name or type name respectively is equal to. In this way, the new name can expand into its definition wherever it is used. For example, we can declare two synonyms for the previously declared names.

```
type unit' : kind := unit.

term •' : unit-again := •.
```

There are three sorts of expressions: *terms*, *types*, and *kinds*. Firstly, **terms** are the data of computation. Such items as numbers, functions, strings, etc. expressible in Func are all terms. There are three forms of terms: *term atoms*, *term functions*, and *term applications*. **Term atoms** are simply names posited as terms. For example: the number `1`, the boolean `true`, the string `"hello world"`, and the boolean function `not` are all term atoms. **Term functions** is a term that encodes a function of one parameter. suppose we want to write a function equivalent to `not` — it takes one boolean parameter, *b*, then evaluates to `false` if *b* is true and `true` if *b* is true. In Func this is written as follows.

```
(b:boolean) ⇒ if b then false else true
```

Names on the left side of the " ⇒ " are the function's *parameters* and the term on the left side is the function's *body*. We could also write the boolean and function of two parameters as follows.

```
(b1:boolean) ⇒
  (b2:boolean) ⇒
    if b1
      then if b2 then true else false
      else false
```

Observe that this function of two parameters is really encoded as a function of one parameter, `b1`, to another function of one parameter, `b2`.[5] We adopt a convention of collecting parameters on the left side of the " ⇒ " and combining their type annotation if they are shared. So we could more simply write the following.

```
(b1 b2 : boolean) ⇒
  if b1
    then if b2 then true else false
    else false
```

[5]It is read this way because the infixed "=>" is right-associative i.e. `a => b => c` implicitly expands to `a => (b => c)`.

However, by itself, a function is inert. In order to *do* something, we need to provide it with arguments. A **term application** is a term that encodes the application of a function to an argument. To write the application of a function `f` to an argument `a`, we write `f a`. For example, to apply the function `not` to the argument `true`, we write `not true`. As for applying functions of multiple parameters, application is left-associative[6] so arguments can simply be written in sequence after the function. For example, to apply the function `and` to two arguments is written

```
and true false
```

which implicitly associates to

```
(and true) false
```

In other words, `and true` evaluates to a function of one parameter, and `false` is provided as that last parameter. This style of writing functions of multiple arguments is called *currying*.

There are two sorts of types that have specially constructed terms that we will be using often: *sum types* and *product types*. Firstly, a **sum type** is a type with terms that must be constructed via a few defined cases. For example, the type `boolean` is a sum type who's terms have two atomic cases `true` and `false`. We can declare such a sum type with the following syntax.

```
type boolean : kind
  { true  : boolean
  | false : boolean }.
```

The " `|` " divides the two cases. Note that the "`boolean : kind`" phrase declares that `boolean` is a type with kind `kind`. The details of types and kinds will be explained in the next section.

We can also use the control structure `cases` in order to break a term of a sum type into to cases. For example, we could declare the function `and` using `cases` as follows.

```
term and (b1 b2 : boolean) : boolean
  := cases b1
       { true  ⇒ cases b2
                   { true  ⇒ true
                   | false ⇒ false }
       | false ⇒ false }.
```

Another useful pattern, especially for sum types, is that of *recursive* definitions i.e. a term who's definition refers to itself. Though this can sometimes result in functions that do not terminate during evaluation, we shall only consider recursive functions that do terminate i.e. *total* functions. A classic *inductive* type is that of the natural numbers, called inductive because it is recursively structured and has a base case. It is a sum type that can be defined as follows.

[6] I.e. `a b c` implicitly expands to `(a b) c`

```
type natural : kind
  { zero : natural
  | succ : natural → natural }.
```

The `zero` case of `natural` encodes the natural number `0`. The `succ` case of `natural` encodes a function that evaluates to the successor of its parameter *n*. We can define a recursive function `add` for natural numbers as follows, where the sum of `zero` and `n` is `n`, and the sum of `succ m'` and `n` is the successor of the sum of `m'` and `n`.

```
term add (m n : natural) : natural
  := cases m
      { zero    ⇒ n
      | succ m' ⇒ succ (add m' n) }.
```

Soon, in section 1.2.3, reduction will be introduced which will allow us to actually carry out the computations encoded by terms such as `add`.

There is a generic type-polymorphic sum type constructor, $\alpha \oplus \beta$, which expresses the sum type of types α and β. The term constructor for the α-case is `left` and for the β-case is `right`.

Secondly, a **product type** is a type with terms that must be constructed using terms of each of a collection of *component* types. For example, the type of pairs of integers, `integral-pair` is the product type of two copies of `integer`, written as follows:

```
type integral-pair : kind
  { x : integer
  ; y : integer }.
```

The two integer components, `x` and `y` are named and separated by a " `;` ". These components names can be used to access the corresponding value of a term of the product type, where `x` and `y` are functions `integral-pair → integer`. For example, if `p` has type `integral-pair`, then `x p` evaluates to the *x*-component and `y p` evaluates to the *y*-component.

There is also a generic type-polymorphic product type constructor, $\alpha \times \beta$. The type constructor for $\alpha \times \beta$ is `(a, b)` where `a` is a term of type α and *b* is a term of type β. Using this constructor for terms of a type product is common in the arguments of functions as well. For example, the following function sums the two components of a `integral-pair`-term.

```
term integral-pair-sum ((x, y) : integral-pair) : integer := x + y.
```

Lastly, there is a parallel correspondence between the syntax of terms and of types. There is a *type atom*, *type function*, and *type application*, which have the same structure of the term versions. However, the term and type syntax levels cannot intermingled, for example a term application cannot evaluate to a type. **Type functions** are used to make a **type-polymorphic** — that is, be generalized over a kind of types. For example, the `list` type is type-polymorphic because it can be applied to a specific types to be the type of lists of integers, the type of lists of booleans, etc. The type-parameter `α:kind` is a sort of "for all `α` of kind `kind`". The type `list` is formed as a recursive type function as follows.

```
type list : kind := (α:kind) ⇒ unit ⊕ (α × list α)
```

where `unit` is the type that has exactly one term of it, •. A list of α is either an empty list, represented by •, or list head α-term and a list tail `(list` α`)`-term. We could also write `list` in the more convenient named sum type notation, where the empty list is named `nil` and constructor appending an α-term to a tail list is named `app`.

```
type list (α:kind) : kind
  { nil : list α
  | app : α → list α → list α }.
```

As `list` is such a common data structure, we shall adopt a few notations for it: `[]` abbreviates `nil` and a`::`as abbreviates `app` a as. Additionally a longer list can be written using the notation `[`a_1`, ..., `a_n`]` which abbreviates a_1`::` ⋯ `::`a_n`::[]`.

Notice that since `list` is parametrized by a type α, the type of `app` is `α ⇒ α → list α → list α`. It can be easy to get confused by the mixture of arrow-like infixed operators. When applying `app`, the argument provided in place for `α` can be provided explicitly or inferred by the other arguments. For example, `app integer 1 (nil integer)` provides `integer` explicitly as a first argument to `app` and `nil` to fill in for `α`, requiring the other arguments to match `α = integer`. However, we could equivalently write `app 1 nil`, since the argument `integer` filling in for `α` is inferred from the fact that `1` is an integer. The latter version will commonly be adopted for the sake of conciseness.

There are a few other data structures that will be introduced along the way in this thesis, and we shall adopt a few other notations. But, ultimately, they all are expressible in terms of the basic syntax presented in this section. See Appendix: Preludes, Func for more detailed examples of basic Func programs.

1.2.2 Typing

Terms and types are related by a **typing judgement**, by which a term is stated to *be of* or *have* a type. The judgement that term a has type α is written as a`:`α. Similarly, types and kinds are related by a **kinding judgement**, where the judgement that type α has kind K is written α`:`K. (Kinding will be much less central to our considerations however, so we refer to typing judgements as also implicitly including kinding judgements.)

A well-formed (following syntax rules) term may or may not be well-typed (following typing rules). Observe that `not true` is well-formed and well-typed, since `not` is a function with a `boolean`-parameter and `true` is a `boolean`-term supplied as the application's argument. Observe also that `not 1` is well-formed but *not* well-typed, since `not` has a `boolean`-parameter yet `1` is an `integer`-term. The checking that terms have properly-corresponding types, as will be detailed in this section, is called **type-checking** or just **typing**.

In order to build typed terms using the generators presented in 7.1, the types of complex terms are inferred from their sub-terms via inference rules within a *judgement context*, where a **judgement context** is a collection of typing (and kinding) judgements. Here, "inference" correlates with the intuitive concept, but the *only* inferences allowed are those exactly corresponding to the collection of inference rules provided — these are pure formalisms. A proposition of the form $\Gamma \vdash a$`:`α asserts that the context Γ entails a`:`α. If a particular judge-

ment J from the judgement context is required in an entailment's premise, then the notation $\Gamma, J \vdash J'$ abbreviates $\Gamma \cup \{J\} \vdash J'$.

With judgements, we now can state the **typing inferences rules** for Func. An inference rule has form

$$\frac{P_1 \quad \cdots \quad P_m}{Q_1 \quad \cdots \quad Q_n}$$

which asserts that the premises $P_1, \ldots, P_m$ entail the conclusions $Q_1, \ldots, Q_n$. For example,

$$\frac{\begin{array}{l}\Gamma \text{ is a judgement context}\\ a \text{ is a term}\\ \alpha \text{ is a term}\\ K \text{ is a kind}\\ \Gamma \vdash a : \alpha\\ \Gamma \vdash \alpha : A\end{array}}{\begin{array}{l}\Gamma \vdash a : \alpha\\ \Gamma \vdash \alpha : A\end{array}}$$

could be a particularly uninteresting inference rule. Explicitly stating the domain of each name in the premises is cumbersome however, so the the domains of names are adopted based on their names:

- Γ is a judgement context,
- $a, b, c, \ldots$ are terms,
- $\alpha, \beta, \gamma, \ldots$ are types,
- $A, B, C, \ldots$ are kinds.

Additionally, unless otherwise stated, a type is presumed to have kind `kind`.

The typing rules for Func are the following.

Table 1.2: Typing in Func

$$\textsc{Simple}\quad \Gamma,\ a{:}\alpha \vdash a{:}\alpha$$

$$\textsc{Simple}\quad \Gamma,\ \alpha{:}A \vdash \alpha{:}A$$

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

$$\textsc{Term-Application}\quad \frac{\Gamma \vdash f\ :\ \alpha \to \beta \qquad \Gamma \vdash a{:}\alpha}{\Gamma \vdash f\ a\ :\ \beta}$$

$$\textsc{Type-Abstraction}\quad \frac{\Gamma,\ \alpha{:}A \vdash \beta{:}B}{\Gamma \vdash (\alpha{:}A) \Rightarrow B\ :\ A \to B}$$

$$\textsc{Type-Application}\quad \frac{\Gamma \vdash \varphi\ :\ \mathsf{A} \to \mathsf{B} \qquad \Gamma \vdash \alpha{:}\mathsf{A}}{\Gamma \vdash \varphi\ \alpha\ :\ B}$$

Though typing rules will be presented for each new syntactical structure in later chapters, in this work they serve as a background assurance rather than a main feature of study. All expressions from here on will be assumed well-typed. For the sake of explanation however, the following demonstrates the typing of the term `[not true]`, which forms a proof that the term is indeed well-typed. Let Γ be the judgement context yielded by the declarations provided so far in this chapter.

$$\frac{\Gamma \vdash \texttt{not : boolean → boolean} \qquad \Gamma \vdash \texttt{true : boolean}}{\Gamma \vdash \texttt{not true : boolean}} \qquad \frac{\Gamma \vdash \texttt{nil : (α:kind) ⇒ list α} \qquad \Gamma \vdash \texttt{boolean : kind}}{\Gamma \vdash \texttt{nil boolean : list boolean}}$$

$$\frac{\Gamma \vdash \texttt{app : (α:kind) ⇒ α → list α → list α} \qquad \Gamma \vdash \texttt{boolean : kind}}{\Gamma \vdash \texttt{app boolean (not true) (nil boolean) : list boolean}}$$

1.2.3 Reduction

Finally, the last step is to introduce **reduction**. So far we have outlined syntax and typing for building well-formed, well-typed expressions in Func, but all these expressions are static. Reduction rules describe how terms can be transformed, step by step, in a way that models computation. The reduction of a term a to a term a' is written $a \twoheadrightarrow a'$. 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." To reduce a term via a series reduction-rule-applications that ends in a value is called **evaluation**.

The following reduction rules are given for Func:

Table 1.3: Reduction in Func

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

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

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

$$\textsc{Split-Left}\quad \texttt{split (left } a\texttt{) } f\ g \twoheadrightarrow f\ a$$

$$\textsc{Split-Right}\quad \texttt{split (right } b\texttt{) } f\ g \twoheadrightarrow f\ b$$

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

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

The most fundamental of these rules is Apply (also known as β-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 $v{:}\alpha$, β-Reduction **substitutes** the input v for the appearances of the function parameter a in the function body b.

For example, the type `boolean` is defined as a type sum.

```
type boolean : kind := unit ⊕ unit.
```

The `left •` case is abbreviated by `true` and the `right •` case is abbreviated by `false`. Then consider the following definition of `not`.

```
term not (b:boolean) : boolean
```

```
  := cases b
      { left  _ ⇒ false   // don't need the unit term parameter,
      | right _ ⇒ true }. // so _ indicates an unused parameter
```

The **cases** structure is actually a notation that the above into the following.

```
term not : boolean → boolean
  := (b:boolean) ⇒ split b (_ ⇒ false) (_ ⇒ true)
```

Then we can reduce an application `not (left •)` as follows.

```
not (left •)
↠
split (left •) (_ ⇒ false) (_ ⇒ true)
↠ (Split-Left)
(_ ⇒ false) •
↠ (Apply)
false
```

As another example, recall the function `add : natural → natural → natural`. The addition of `N2 := succ (succ zero)` to `N3 := succ (succ (succ zero))` is computed as follows.

```
add N2 N3
↠ (Definition)
(m n ⇒ cases m{ zero ⇒ n | succ m' ⇒ succ (add m' n) })
  (succ(succ zero)) N3
↠ (Apply)
cases (succ(succ zero)){ zero ⇒ N3 | succ m' ⇒ succ (add m' N3) }
↠ (Split-Right)
(m' ⇒ succ(add m' N3)) (succ zero)
↠ (Definition)
(m' ⇒ succ((m n ⇒ cases m{zero ⇒ n | succ m' ⇒ succ(add m' n)}) m' N3))
  (succ zero)
↠ (Apply)
succ((m n ⇒ cases m{ zero ⇒ n | succ m' ⇒ succ(add m' n) }) (succ zero) N3
   )
↠ (Apply)
succ(cases (succ zero){ zero ⇒ N3 | succ m' ⇒ succ(add m' N3) })
↠ (Split-Right)
succ(m' ⇒ succ(add m' N3) zero)
↠ (Definition)
succ(succ((m n ⇒ cases m{ ⇒zeron | succ m' ⇒ succ (add m' n) }) zero N3))
↠ (Apply)
succ(succ((cases zero{ zero ⇒ N3 | succ m' ⇒ succ (add m' N3) })))
↠ (Split-Left)
```

```
succ(succ((_ ⇒ N3) •)) // the • comes from zero = left •
↠
succ(succ N3)
=:
N5
```

1.2.4 Properties

With the syntax, typing, and reduction defined for Func, we now have a completed definition of the language. However, some of the design decisions may seem arbitrary. This particular framework is good because it maintains a few nice properties that make reasoning about Func intuitive and extendable. Here when referring to a *term* there is the implicit premise that the term is well-formed.

Theorem 1.1. (Type-Preserving Substitution in Func). If $\Gamma, a{:}\alpha \vdash b{:}\beta$, $\Gamma \vdash v{:}\alpha$, and value v, then $\Gamma \vdash ([v/a]b){:}\beta$. I.e. the Apply reduction rule always results in a term with the type of the function's output.

Theorem 1.2. (Reduction Progress in Func). If $\{\} \vdash a{:}\alpha$, then either value a or there exists a term a' such that $a \twoheadrightarrow a'$. I.e. well-typed terms are either values or reducible.

Theorem 1.3. (Type Preservation in Func). If $\Gamma \vdash a{:}\alpha$ and $a \twoheadrightarrow a'$, then $\Gamma \vdash a'{:}\alpha$. I.e. reducing a well-typed term always results in a well-typed term of the same type.

Theorem 1.4. (Strong Normalization in Func). For any term a, either value a or there is a sequence of reductions from a that ends in a term a' such that value a'. I.e. every term evaluates in a finite number of steps.

Chapter 2

Imperative Effects

2.1 Declarative and Imperative Languages

Previously in section 1.2, we defined Func as a basic variant of the λ-calculus and contrasted it to a Turing-machine inspiration for programming languages. This distinction arises in programming languages in general as between *declarative* (generally λ-calculus-like) and *imperative* (generally TM-inspired sequentially-instructional) languages. *Declarativ*ed 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.

> **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 (unsafe) stateful and stateless expressions. 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.

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 (declaratively)**. 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 natural-number variable, and initialize i to be 0. If i is less 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.

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 (via the "repeat this sentence" clause) that repeats for each list element and a variable to store the partial sum.

However, the categories of imperative and declarative are not so strict — many languages borrow styles 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.

2.2 Computation with Effects

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. For terms in Func, reductions are **context independent** — the evaluation of a term relies only on information contained explicitly within the term.

In imperative languages, and so almost all used languages, it is rarely the case that evaluation is context independent. The real state of programming is full of stateful and contextualized structures, such as variables, pointers, object instances, and many more. As a simple example, just consider mutability. This Java program defines a class with two

[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.

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.

```
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
}
```

[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.

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 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 Func, 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, Func 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 Func programs exist purely as mathematical structures. However, this aspect also yields a language that is inert relative to the physical world; Func 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

[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).

[6]It could be argued that there are in fact implicit contexts involved in specific implementations of Func, since those implementations have to use computer memory to keep track of the data used in a Func 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

Func, 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 Func 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 Func (or an untyped but similarly-pure language, Lisp) for anything useful was rarely entertained. The rise of object-oriented programming has also proliferated the use of implicit contexts, since each object instance acts as a new user-specified (via object-classes) implicit context.

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 Func at all if it is so practically irrelevant?

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.

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

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*.

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.

> An expression 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: *mutability*, *exception*, *nondeterminism*, and *input/ouput*. They are detailed in the following paragraphs.

[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.

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.

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.

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.

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.

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.

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 the previous definition of "effect," 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) Freely mix 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 normal scopes.

A concept important to the consideration of this dilemma is that of "safety." The **safety** of a programming language is a measure of how much and in what ways it allows and promotes a correspondence between the *intended*, *apparent* behavior and the *actual* behavior of its programs. In other words, a language is **safe** if it is *easy* to accurately and precisely predict the behavior of its programs by analyzing code source code, and a language is **dangerous** if it is *hard* to accurately and precisely predict the behavior of its programs by analyzing their source code.[11] The Dilemma of Purity and Effect highlights how effects can be a source of danger. Many proponents of horn 1 claim that purity allows significantly improves the safety of a language, since the factors relevant to a program's execution are kept explicit and thus are easier to reason about. A main form of this is referred to as **type-safety**; an expressive type system can require programs to make explicit and handle potential dangers in order to type-check (part of the process of analyzing code).

Almost all languages fall decisively on the side of horn 1, but are there ways to salvage some of the safety from horn 2 (in particular, type-safety)? In this and the following chapters, we will follow an arc of declarative language design that incrementally builds the a middle way between the two horns, by introducing structures and type-systems that extend a horn 1 perspective to allowing some horn-2-looking behavior. First we shall start with a language, Impe, that in a simple way extends the declarative language Func with imperative-style forms of the previously-described example effects.

[11] Difficulty can be measured on a scale from *easiest* to *hardest* based on some accepted constraints (which generally all correlate very positively). For example: allowed computational power, weighting of accuracy, weighting of precision, allowed length of program's code, allowed time for program to work, etc.

2.3 Language Impe

Language Impe is a first example of implementing effects by extending Func. Its approach is inspired by the language Standard ML, which is commonly used in teaching programming languages theory (especially the declarative style). The strategy is to realize the implicit/-explicit context distinction very strictly — a selection of primitive terms are introduced that are designated to interact with the implicit context in a way not able to be referenced within Impe code. Since the resulting language style closely mimics imperative languages' usual look, call this effect implementation **imperative effects**. Note importantly, though, that these dubbed imperative effects are implemented within a declarative language (namely, Impe).

This section introducing Impe describes its implementation of the example effects given in the section 2.2.1, but keep in mind that the implementation strategy is meant to apply to effects in general.

2.3.1 Imperative Effects

As imperative effects mimic imperative languages, we would like to write code that executes sequentially and with state. Recall the task of summing a list of integers from section 2.1. A function that does this task can be written in Func as the following:

Listing 2.1: Sum a list of integers (declaratively)

```
term sum-list-integer (l : list integer) : integer
  := cases l
       { []     ⇒ 0
       ; i :: l' ⇒ i + sum-list-integer l' }.
```

In the imperative language Java, a function that does this task can be written as the following:

Listing 2.2: Sum a list of integers (imperatively in Java)

```
int sumListInt(List<int> l)
  {
    int s = 0;
    int i = 0;
    while i < l.length()
      {
        x = l[i];
        s = s + x;
        i = i + 1;
      }
    return s;
  }
```

So how might this same, imperative function be written in the declarative language Impe? Consider the following:

Listing 2.3: Sum a list of integers (imperatively in Impe)

```
term sum-list-integer (l : list integer) : integer
  := do
       { let s : mutable integer := initialize 0 in
         let i : mutable natural := initialize zero in
         while !i < length is
           loop do
             { let x : integer := index l !i in
               s ← !s + x
             ; i ← !i + 1
             ; iterate • }
           done !s }
```

Notice the parallels between this function and the Java function. This example makes use of the sequencing and mutability effects, all of which and more will be detailed in the following sections.

Reduction contexts. The central concept introduced for Impe is the **reduction context** — an implicit context present during the reduction of terms. We symbolize a reduction context with Δ, and the presence of Δ during the reduction of a term a is written

$$\Delta \parallel a$$

where the $\parallel$ separates the implicit context on the left-side from the explicit context on the right side. We modify the SIMPLIFY rules for Impe to account for the reduction context as follows:

Table 2.1: Reduction in Impe

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

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

By these rules, impure reductions in sub-terms also affect the implicit context of the outer terms, and so Δ is treated as globally-accessible. The Δ is carried around as the state of the program during reduction, yielding the stateful programming characteristic of the imperative style.

The strategy of Impe's implementation of effects is introduce a reduction context particular to each effect, and then have reduction rules that use each reduction context to perform

the effect. The Δ used above stands in for the total of all these reduction contexts considered together.

2.3.1.1 Sequencing

The sequencing effect directs a sequence of terms to have their effects performed in a the sequence's order. To provide this effect, Impe declares a primitive term `sequence` parametrized by two terms and encodes the sequencing of their effects.

Listing 2.4: Primitive for sequencing

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

Listing 2.5: Notations for sequencing

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

do{ «term»₁ ; ⋯ ; «term»ₙ }
  ::=
    «term»₁ >> ⋯ >> «term»ₙ

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

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

A term of the form `sequence a b` encodes the performance of the effects of `a`, and then the performance of the effects of `b`. As a default, the total term reduces to the result of `b`. The following reduction rules specify this behavior.

Table 2.2: Reduction in Impe: Sequencing

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

$$\textsc{Next}\quad \frac{\text{value } v}{\Delta \parallel v \gg a \twoheadrightarrow \Delta \parallel a}$$

2.3.1.2 Mutability

A simple way of introducing mutability to a declarative language like Func is to posit a primitive type `mutable`, which is parametrized by a type α, as the type of mutable α-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` α 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 2.6: Primitives for mutablility

```
primitive type mutable : kind → kind.

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

Listing 2.7: Notations for mutability.

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

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

A term of type `mutable` α is a sort of reference to a store where the an α-term is stored. We introduce a reduction context $\mathcal{S}$ that contains such stores, and write $\mathcal{S}[\![id \mapsto v]\!]$ to indicate that $\mathcal{S}$ contains a store referenced by id and currently stores v. Additionally, in order to create fresh stores, write $\mathsf{new}(\mathcal{S}) \rightsquigarrow (\mathcal{S}', id)$ to indicate that $\mathcal{S}$ can create a fresh store that is referenced by id in the new context $\mathcal{S}$. From this setup of $\mathcal{S}$, the following result as the reductions for performing the mutability effect.

Table 2.3: Reduction in Impe: Mutability

$$\textsc{Initialize}\quad \frac{\text{value } v \qquad \mathsf{new}(\mathcal{S}) \rightsquigarrow (\mathcal{S}', id)}{\mathcal{S} \parallel \texttt{initialize}\ v \twoheadrightarrow \mathcal{S}'[\![id \mapsto v]\!] \parallel id}$$

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

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

Note that the omission of other reduction contexts — such as Δ and the contexts to be introduced in the following sections — implies that they are unchanged by the reduction rule.

Example. Suppose we want to update the values of two integer variables so their values are ordered. The following function implements this using the mutability effect.

```
term sort-two (x y : mutable integer) : unit
  := if !x > !y
       then do
         { let i := !x in
           x ← !y
         ; y ← i  }
       else •.
```

The following demonstrates how an application of this function reduces. The integer variables give are initialized such that $id_x \mapsto 4$ and $id_y \mapsto 2$, and after applying `sort-two` they are modified such that $id_x \mapsto 2$ and $id_y \mapsto 4$.

$\mathcal{S}[]$ ‖ `sort-two (initialize 2) (initialize 4)`
(Initialize x2)
↠ $\mathcal{S}[id_x \mapsto 4, id_y \mapsto 2]$ ‖ `sort-two` id_x id_y
(Definition)
↠ $\mathcal{S}[id_x \mapsto 4, id_y \mapsto 2]$ ‖ `if !`id_x `>` `!`id_y
`then let i := !`id_x `in (`id_x `← !`id_y `>> `id_y `← i)`
`else •`
(Get)
↠ $\mathcal{S}[id_x \mapsto 4, id_y \mapsto 2]$ ‖ `if 4 > 2`
`then let i := !`id_x `in (`id_x `← !`id_y `>> `id_y `← i)`
`else •`
(Simplify)
↠ $\mathcal{S}[id_x \mapsto 4, id_y \mapsto 2]$ ‖ `let i := !`id_x `in (`id_x `← !`id_y `>> `id_y `← i)`
(Get)
↠ $\mathcal{S}[id_x \mapsto 4, id_y \mapsto 2]$ ‖ id_x `← !`id_y `>> `id_y `← 4`
(Get)
↠ $\mathcal{S}[id_x \mapsto 4, id_y \mapsto 2]$ ‖ id_x `← 2 >> `id_y `← 4`
(Set)
↠ $\mathcal{S}[id_x \mapsto 4, id_y \mapsto 2]$ ‖ `• >> `id_y `← 4`
(Next)
↠ $\mathcal{S}[id_x \mapsto 2, id_y \mapsto 2]$ ‖ id_y `← 4`
(Set)
↠ $\mathcal{S}[id_x \mapsto 2, id_y \mapsto 4]$ ‖ •

2.3.1.3 Exception

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 Func with naive exceptions is that it requires exceptions to be of the same type as the expected result. Schematically, our safe division function example looks like this:

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

Suppose that we do not have exceptions, like in Func. Then in the definition of `divide-safely`, 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-of` α of which terms are each a label for exceptions parametrized by α. Such exception-labels should only be constructed primitively, so for the creating of new terms of type `exception-of` α we introduce a new declaration **`exception`** *e* **`of`** α, which declares an exception term *e* of type `exception-of` α. These structures are captured by the following syntax rules:

Table 2.4: Syntax for Impe: Exception

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

Instances of `exception-of` α (which must be introduced primitively) are specific exceptions that are parametrized by a term of type α. This allows exceptions to store some data, perhaps about the input that caused their throw. The term `throw` throws an exception, requiring its α-input, and evaluating to any β-result. The term `catching` takes a continuation

[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.

of type `(exception-of α → α → β)` for handling any α-exceptions while evaluating a given term of type β. 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." The primitive declarations for `throw` and `catching` are given, along with a useful notation for `catching`.

Listing 2.8: Definitions for exception

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

Listing 2.9: 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
```

Finally, for the reduction rules for exception. Let us introduce a new reduction context $\mathcal{E}$ to manage exception. If no exception has been raised, then write $\mathcal{E}[]$. If an exception e has been thrown with argument x, then write $\mathcal{E}[e, x]$. Additionally, since the result of throwing an exception is an invalid value, we shall denote it by $\lightning$. Then the reduction rules are as follows.

Table 2.5: Reduction in Impe: Exception

$$\textsc{Throw}\quad \frac{\text{value } x}{\mathcal{E}[]\ \|\ \texttt{throw}\ e\ x \twoheadrightarrow \mathcal{E}[e,x]\ \|\ \eighthnote}$$

$$\textsc{Catch}\quad \mathcal{E}[e,x]\ \|\ \texttt{catch}\ \{\ e\ a \Rightarrow b\ \}\ \texttt{in}\ \eighthnote \twoheadrightarrow \mathcal{E}[]\ \|\ (a \Rightarrow b)\ x$$

$$\textsc{Valid}\quad \frac{\text{value } v}{\mathcal{E}[]\ \|\ \texttt{catch}\ \{\ e\ a \Rightarrow b\ \}\ \texttt{in}\ v \twoheadrightarrow \mathcal{E}[]\ \|\ v}$$

$$\textsc{Raise}\quad \mathcal{E}[e,x]\ \|\ \texttt{a}\ \eighthnote \twoheadrightarrow \mathcal{E}[e,x]\ \|\ \eighthnote$$

$$\textsc{Raise}\quad \mathcal{E}[e,x]\ \|\ \eighthnote\ \texttt{b} \twoheadrightarrow \mathcal{E}[e,x]\ \|\ \eighthnote$$

Example. With this exception framework, we can concretely write the divide-safely function via the following.

```
1 exception divide-safely-by-0 of integer.
2
3 term divide-safely (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 `divide-safely` 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 `divide-safely` 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.

2.3.1.4 Nondeterminism

For our consideration here, a basic nondeterministic computation is defined to have a range of values, given by a list, from which a random result is chosen. The primitive term `sample` encodes this effect.

Listing 2.10: Primitive for nondeterminism

```
primitive term sample (α : Type) : list α → α.
```

For reduction, `sample` will simply appeal to an external interface $\mathcal{ND}$ to pick on of the options to result in. Such an appeal is written $\mathcal{ND}$`(sample` l`)` where l is the list of options.

Table 2.6: Reduction in Impe: Nondeterminism

$$\textsc{Sample}\quad \mathcal{ND} \parallel \texttt{sample}\ l \twoheadrightarrow \mathcal{ND} \parallel \mathcal{ND}\texttt{(sample}\ l\texttt{)}$$

Specification of $\mathcal{ND}$. Of course, what an appeal $\mathcal{ND}$`(sample` l`)` results in is dictated by the specific implementation of $\mathcal{ND}$. The type requirement is that $\mathcal{ND}$`(sample` l`)` yields a value of the type of the elements of l. But other than this, the specification of $\mathcal{ND}$ is left external to Impe.

Example. Suppose we would like to flip a coin. We can use the `boolean` type as the type of results, where `true` corresponds to "heads" and `false` corresponds to "tails." Then a function that flips a coin nondeterministically can be written as follows:

```
term flip-coin (_:unit) : boolean := sample [true, false].
```

2.3.1.5 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 2.11: 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`.

Table 2.7: Reduction in Impe: I/O

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

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

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 Impe'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 Impe's semantics; $\mathcal{IO}$ is a black box from the point of view within Impe. An external implementation of $\mathcal{IO}$ that could be compatible with Impe must satisfy the following specifications:

Specification of $\mathcal{IO}$

- $\mathcal{IO}$`(input •)` resolves as a value of type `string`,
- $\mathcal{IO}$`(output s)` resolves as •.

I use the term "resolve" here to emphasize that $\mathcal{IO}$'s capabilities are operating outside of the usual semantics of Impe in order to evaluate.

At this point, we can express the a friendly greeting program.

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

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

Implementation 1 of $\mathcal{IO}$:

- $\mathcal{IO}$`(input •)`:
  1. Prompt the console for user text input.
  2. Interpret the user text input as a string, then resolve as the string.
- $\mathcal{IO}$`(output s)`:
  1. Write `s` to the console.
  2. Resolve as •.

As intended by Impe's design, this implementation will facilitate `greetings` appropriately: the user will be prompted and then a greeting to their input will be printed. Beyond the requirements enumerated by the specification of $\mathcal{IO}$ however, Impe does not guarantee anything about how $\mathcal{IO}$ behaves. For example consider the following alternative implementation 2 of $\mathcal{IO}$ that still meets the specification.

Implementation 2 of $\mathcal{IO}$:

- $\mathcal{IO}$(input •):
  1. Set the toaster periphery's mode to *currently toasting*.
  2. Resolve as a string representation of the toaster's current temperature.
- $\mathcal{IO}$(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 intent of Impe, though unfortunately it is still compatible with $\mathcal{IO}$'s specification. In the way that Impe 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 $\mathcal{IO}$. The I/O effect is very dangerous (to use the dichotomy between *safe* and *dangerous* from 2.2.1).

2.4 Motivations

This chapter has introduced the concept of effects in programming languages, and presented Impe as a simple way to extend a basic lambda calculus, Func, with a sample of effects. For each effect, the implementation strategy used is to introduce a facilitating reduction context: $\mathcal{S}, \mathcal{E}, \mathcal{IO}, \mathcal{ND}$. Each these reduction contexts are examples of implicit contexts since they are not directly able to be referenced from within Impe— 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 $\mathcal{ND}$. 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 Impe-values, and the context $\mathcal{E}$ is either empty or contains a pair of an Impe-exception-name and a Impe-value. While these reduction contexts may be implemented in a variety of ways while still meeting Impe's specification for them, each of these implementations will have the same behavior from the point of view of reduction in Impe.

Considering group (2), $\mathcal{IO}$ and $\mathcal{ND}$ 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 $\mathcal{ND}$ is a black-box interface that can produce a random float.

A reason external effects are introduced to Impe this way is because it is infeasible to introduce an entire model of their Impe-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 Impe. 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 Impe's specification for it. The same applies to $\mathcal{ND}$. 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 $\mathcal{ND}$ 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 Impe 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 Func. 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 Impe, there are a couple inspirations for such improvement to take away.

Generalized context for effects. In Impe, 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 Impe 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.

Explicitly-typed effects. In Impe, 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

[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*), modularity (pure code can be used anywhere and have the same behavior, impure code can be sometimes be combined), and optimization (impure activities that take more resources to do sporadically can be reorganized around pure code to improve efficiency, without changing behavior). These considerations are beyond the scope of this work, but provide a context for why these formal analyzability is desirable.

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.

Programmable internal effects. In Impe, 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*. Monadic effects will address these inspirations and, in doing so, refine our goals for implementing effects.

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.

Chapter 3

Monadic Effects

3.1 Introduction to Monads

The concept of *monad* was first introduced in category theory, but since been relevant to theoretical computer science as well (particularly to type theory). A notion of monads is presented as a novel way to introduce effects into declarative, λ-calculus-inspired languages by [8] and [11]. It turns out that monads can be used as a general framework for internally implementing explicitly-typed effects in an extension of Func. This chapter will demonstrate by developing such a language, Mona, in a relatively programmer-friendly way.

To start, take the division of the implicit context from the explicit context so fundamental to Impe's structure. We wrote $\Delta \parallel a$ to represent a term a with implicit context Δ. Suppose that a has an effect that appeals to Δ. From here on, refer to an impure term (such as a) as a **computation**. A computation has two parts: the implicit context on which it relies (e.g. the left side of "$\parallel$"), and the explicit context in which its result is evaluated (e.g. the right side of "$\parallel$"). Call a computation that results in a term of type α an **α-computation**, and call α the **result type** of the computation. For example, the Impe term `coin-flip •` is a `boolean`-computation of the nondeterministic effect.

As mentioned earlier, Mona's effects are internal and explicitly-typed. In Impe, the computation `coin-flip •` had type `boolean` just like a pure term such as `true` or `false` would. To make effects explicitly-typed, Mona will introduce a typing scheme that yields a term such as `coin-flip •` to instead have a type like `nondeterministic boolean`, where the type `nondeterministic` indicates a computation using the nondeterministic effect and is parametrized by its result type. However, a result of this is that a term with a type of the form `nondeterministic` α cannot be treated just like any other term with type α. This raises a problem because we would like to write functions and other expressions that make use of these computations and their results. For example, we might want to apply a function f `:` $\alpha \to \beta$ to the result of a `(nondeterministic` α`)`-term. But there are restrictions on the sorts of things we should be able to do, since the indication that `nondeterministic` α is a computation and not just a pure value must be preserved in order for the indication to consistently reflect the purity of impurity of terms. With the example, in applying f to

a term `m : nondeterministic α`, we expect the result to be a `β`-computation that uses the nondeterminism effect in the way defined by `m`, and then applies `f` to the result, which should have the type `nondeterministic β`.

In other words, there is a selection of general *capabilities* we expect to be provided by explicitly-typed computations, so that they can be used with other terms to write the same programs as able to be written in Impe. In the example, a term that provides the capability of applying `f` to the result of `m` has type `(α → β) → nondeterministic α → nondeterministic β`. By using explicit types to indicate effects, we can formalize the idea of "capability" in terms of types.

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

The mentioned term with type `(α → β) → nondeterministic α → nondeterministic β` provides a capability for the type `nondeterministic`.[1] We shall formulate a qualification for effects in Mona in terms of capabilities required to be provided by **effect types** (such as `nondeterminstic`). The qualification for monads provide just such a qualification for this implementation of effects.

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 in Impe by the introduction of a globally-accessible, store of mutable variables as the reduction context `S`. This implementation heavily relied on the reduction context, along with the associated reduction rules, so manage this effect — all of which is implicit and not directly-accessible to a programmer. The rules governing mutability are not expressed within Impe code itself, and so are a source of potential danger. So, is there a way to implement mutability in a way more explicitly expressible within Func?

This chapter explores a certain way to accomplish this. However, since Func is pure, such an implementation cannot provide *true* mutability since Func terms are definitionally immutable. The abstraction necessary is to think of mutability in more expressively-typed terms. Refer to a computation that uses the mutability effect with *state type* `σ` and result type `α` as a **`σ`-stateful** `α`-computation. The **state type** of a computation using mutability is the type of the implicit mutable state kept track of during its evaluation.

We can model mutability in Func types as a function from the initial state to the modified state:

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

[1] Implicitly, the term is not considered to provide a capability for either `α` or `β` since they abstracted from the type.

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 (i.e. possibly-mutated) 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 Impe's implementation of this effect, it posited two primitive terms: `get` and `set`. Using `mutable` we can define these terms in Func as:

```
term get (σ : kind)       : mutable σ σ    := s ⇒ (s, s).
term set (σ : kind) (s:σ) : mutable σ unit := _ ⇒ (s', •).
```

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

As for the capabilities expected of this type in order to consider it an effect, we shall consider *sequencing*, *binding*, *lifting*, and *mapping*.

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 sequence (σ α β : kind)
  : 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 `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 σ β` 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 bind** (or just **binding**), as it binds to the α-result of m the α-parameter of fm.

```
term bind (σ α β : kind)
  : 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 `sequence` and `bind` is superfluous i.e. can be defined in terms of the other. Consider the following re-definition of `sequence`:

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

This construction demonstrates how `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 state-relevant operations needed to express stateful computations in Func. The key difference between Impe and our new Func-implementation of the mutable effect is that Impe treats stateful computations just like any other kind of pure computation, whereas Func "wraps" σ-stateful α-computations using the type `mutable σ α` rather than just an "unwrapped" type α. 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*.

Lifting. A (relatively[2]) 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 $\bullet$. The missing expression we need is a function that results in a value without touching the state at all. Call this function `lift`, which should have the type `α → mutable σ α`.

```
term lift (α : kind) : α → 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
  := bind get
```

[2]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 Appendix: Preludes, Mona).

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

Mapping. A relatively pure function can be be *mapped*[3] over stateful computation results by applying it to the result of the computation and then lifting. In other words, to lift[4] a function of type `α → β` to a function of type `mutable σ α → mutable σ β`. Call this function `map`.

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

As seen above, `mutable`'s `map` implementation arises very straightforwardly from `mutable`'s `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 (as will be demonstrated in section 3.4).

Example. The following implements the `sort-two` program from section 2.3.1.2.

```
type state : kind { x:integer; y:integer }.

term get-x : mutable state integer := bind get x.
term get-y : mutable state integer := bind get y.

term modify (f : σ → σ) : mutable σ α → mutable σ α
  := bind get (set ∘ f).

term set-x : integer → mutable state integer := modify set-x.
term set-y : integer → mutable state integer := modify set-y.

term sort-two : mutable state unit
  := bind get-x (x ⇒
       bind get-y (y ⇒
         if x > y
```

[3]The terms *map, mapped, mapping* are very much overloaded between computer science and mathematics terminology. Here, to **map** a function over a data structure is to apply the function to the contents (in some relevant way) of the data structure. For example, mapping a function `f` over a list `[1, 2, 3]` reduces to `[f 1, f 2, f 3]`.

[4]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*.

```
        then sequence (set-x y) (set-y x)
        else lift •)).
```

The utility functions `get-x`, `get-y`, `set-x`, and `set-y` are declared in order to manage a state type with multiple components. As demonstrated, the state type `variables` is defined as a named product with two components `x` and `y`, corresponding to the two variables from Impe's implementation of `sort-two`. This pattern can be generalized to host a mutable state with any number of named components by using a named product type `state` of the components, and additionally defining the matching `get-`*name* and `set-`*name* capabilities of `mutable state` for each component *name*.[5]

Considering such an example as this, another question becomes apparent: how would a term of type `mutable σ α` be "run" in order to extract its final state and result? The way that `mutable` is defined, as `mutable σ α = (σ → σ × α)`, contains the answer: a term of type `mutable σ α` is a function waiting for an initial state of type `σ` in order to produce the desired final state and result. So, to "run" a term using the mutability effect is just to provide the initial state. The following function implements an intuitive interface for this.

```
term initialize (σ α : kind) (s:σ) (m : mutable σ α) : α
  := m s.
```

With the `sort-two` example, we can use `initialize` to run the effect on an initial state.

```
initialize (2, 1) sort-two   ↠   (1, 2)
```

3.1.2 Formalization of Monad

In section 3.1.1, we implemented a variety of terms specifically for the mutability monad. To summarize, they were: `get`, `set`, `sequence`, `bind`, `lift`, and `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 `sequence` provides the monad-essential capability of sequencing, but was later discovered to be expressible in terms of `bind` without reference

[5]The get and set functions defined for each component of `state` follow directly from the definition of the named product as functions that access or modify There is clearly a lot of redundancy in this pattern, which suggests that there is some more general way of deriving get, set, and other similar functions from named products. A common generalization is that of *lenses*, which can be neatly made to work for both the immutable context (e.g. for named products in Func) and generically lifted to the mutable context as implemented by the monadic mutability effect. In the Haskell programming language, such lenses are implemented in both contexts (including for the mutable context by using what Haskell calls the `State` monad) by the `lens` package: https://hackage.haskell.org/package/lens.

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 `lift` and `map` provide the monad–essential capabilities of lifting and mapping. Though `map` was implemented using `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 Func 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 Func?

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 Func with monads, let us first take a detour on *type-classes*, of which monad is just one sort.

3.2.1 Object-Oriented 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 Func (see Appendix: Preludes, Func), though `IntegerList` appears relatively clunky in Java.

In relating object-classes to Func'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 Func 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 Func.

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[6] 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:

[6] 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.

```
class Cat extends Animal {

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

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

```
// type—class
type Animal (α : kind)
  := α → unit → unit // eat
  × α → unit → unit // sleep
```

A term of type `Animal α` for some type α 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 : kind) : kind
  := (α → string)  // name
  × (α → string)  // eat
  × (α → string). // sleep

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

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.

Type-class Instantiation. Instantiating the a type α 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. Note that as type classes are given here (which is how they are typically given in the real-world programming languages that offer type-classes), a term can be instantiated of a type class only once.

Bringing it all together, we can rewrite the previous program with the type-class `Animal`, its instances `cat` and `dog`, and the function that uses type-class `Animal` membership as follows:

```
type Animal (α:kind) : kind
  { name  : α → string
  ; eat   : α → string
  ; sleep : α → string }.

type cat := string.

instance Animal cat
  { name  c := "cat"
  ; eat   c := name c ++ " eats kibble."
  ; sleep c := name c ++ " naps." }.

type dog := string.

instance Animal dog
  { name  d := "dog"
  ; eat   d := name d ++ " eats steak."
  ; sleep d := name d ++ " sleeps wrestlessly." }

term simulate-Animal (α:kind) {Animal α} (a:α) (steps:natural) : string
  := (string-concat ∘ repeat steps)
       (eat a ++ " " ++ sleep a ++ " ").
```

Notice that the function `simulate-Animal` requires that α is an instance of the type-class `Animal` via the argument `{Animal α}`. This is special kind of **implicit argument**, which need not be provided when applying `simulate-Animal` but is type-checked by looking for the appropriate **`instance`** `Animal α` declaration somewhere is the program. We can apply `simulate-Animal` as simply as the following,

```
simulate-Animal ("Labby":dog) 3
↠ ⋯ ↠
"Labby eats steak. Labby sleeps wrestlessly. Labby eats steak. Labby
   sleeps wrestlessly. Labby eats steak. Labby sleeps wrestlessly. "
```

where the `Animal`-instance of `dog` is passed implicitly.

3.3 Monad

3.3.1 Monad Type-Class

It turns out that we can model the Monad type-class as a parametrized type, with the type `Monad (M : kind → kind) : kind` 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 3.1.2.

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 : kind → kind) : kind
  { lift : (α : kind)   ⇒ α → M α
  ; map  : (α β : kind) ⇒ (α → β) → M α → M β
  ; bind : (α β : kind) ⇒ 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 ; let «term-param»* ← «term»* ; ⟦ «term» ; ⟧2 }
  ::=
    do{ ⟦ «term» ; ⟧1 ; let «term-param»* ← «term»* in do{ ⟦ «term» ; ⟧2 } }
```

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

Example. The following implements the `sort-two` program, which was already adapted to Mona via the Example paragraph in section 3.1.1, but here makes use of the new monadic notation.

```
type state : kind { x:integer; y:integer }.

term get-x : mutable state integer := get >>= (lift ∘ x).
term get-y : mutable state integer := get >>= (lift ∘ y).

term modify (f : σ → σ) : mutable σ α → mutable σ α
  := get >>= (set ∘ f).

term set-x : integer → mutable state integer := modify set-x.
term set-y : integer → mutable state integer := modify set-y.

term sort-two : mutable state unit
  := do
      { vx ← get-x
      ; vy ← get-y
      ; if vx > vy
          then set-x y >> set-y x
          else lift • }.
```

The expected `initialize` function implemented exactly as before in section 3.1.1.

Here, in particular, the `>>=` infixed notation for `bind` and the `do` notation for sequencing make the code much more readable. The `do` block is very reminiscent of an imperative style.

3.3.2 Mondad Properties

We have described a type-class for monad that requires a selection of capabilities in order to be instantiated for particular monads. However, there is more to being a monad than just implementing these capabilities — in particular the implementations must satisfy what are called the *coherence conditions* for monad capabilities.[7]

Table 3.1: Coherence conditions for `Monad`

Let M be a monad instance and α, β be types.

- Extending lift yields the identity on $M\ \alpha$, written as `extend lift = identity`.
- For any functions $f : \alpha \to M\ \beta$ and $g : \beta \to M\ \gamma$, we have `extend g ∘ extend f = extend (extend g ∘ f)`
- For any function $f : \alpha \to M\ \beta$, we have `extend f ∘ lift = f`

[8]

The conditions make use of the following functions:

```
term join {Monad M} (α:Type) (mm : M (M α)) : M α
  := mm >>= identity.

term extend {Monad M} (f : α → M β) : M α → M β
  := join ∘ (lift ∘ f)
```

In fact, we could have defined `lift` and `map` in terms of `join` and `extend`, but it turns out that `lift` and `map` are much more intuitive from a programing-languages perspective and the conditions presented are from a category-theoretic perspective. Note that the coherence conditions are not enforced by Mona, but they are assumed checked throughout the rest of this work whenever a monad is instantiated.

[7]The conditions presented here for monads are in a form more directly related to what are called Kleisli triples. However the resulting structures are equivalent.

[8]These conditions make use of a meta-level equivalence relation $=$, which is not expressible withing Mona (nor any other language presented in this work). Two terms are considered equal under this relation if and only if they are the same value or, if they are functions, the reduce to the same value when given the same inputs (this concept becomes more complicated when implicit contexts are involved, which Mona mostly avoids). This is the basic idea of *functional extensionality*, where the proposition $f = g$ claims that f and g are *functionally equivalent*. See [1] and the later [14] for an introduction to the type-theoretic treatment of this subject.

3.4 Language Mona

So far we have seen the mutability effect represented as a monad, which is entirely representable in Func code. We shall define a new language, Mona, that uses the monadic strategy for implementing the mutability effect to implement effects in general. This is very similar to how monadic effects are implemented in the Haskell programming language.

3.4.1 Monadic Internal Effects

Constructively-instantiated monads (as apposed to primtively-instantiated) explicitly define the behavior of their corresponding effects, and so these are internal effects. Recall how in Impe, each internal effect had a fully-mathematically specified reduction context. In Mona, the monad instance for each internal effect will play the role that each Impe reduction context played.

3.4.1.1 Exception

In Impe, 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 : kind) : kind
  := 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 : kind) ⇒ Monad (exceptional ε)
  { map (α β : kind)
      (f : α → β) (m : exceptional ε α)
      : exceptional ε β
      := cases m
           { throw e ⇒ throw e
           | valid a ⇒ valid (f a) }
  ; lift (α : kind)
      (a : α)
      : exceptional ε α
      := valid a
  ; bind (α β : kind)
      (m : exceptional ε α) (fm : α → exceptional ε β)
      : exceptional ε β
      := cases m
           { throw e ⇒ throw e
           | valid a ⇒ fm a } }.
```

Catching exceptions. For terms of type `exceptional ε`, we would like a catching-capability such as appeared in Impe's syntax for exception. Such a capability takes an input `(exceptional ε α)`-term and outputs a pure `α`-term, given a exception-continuation of type `ε → α`. The term `catching` implements this capability, and the notation following it allows catching to be written in a way reminiscent of Impe but now explicitly typed.

Listing 3.9: Definition of catching for exceptions

```
term catching (ε α : kind) (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-param»_1 : «type»_1) ⇒ («term»_2 : «type»_2) } in «term»_3
  ::=
    catching «type»_1 «type»_2 ((«term-param»_1 : «type»_1) ⇒ («term»_2 : «type»_2))
      «term»_3
```

Notice that we don't need to specify anything analogous to «*exception-name*» from Impe'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 α would have the type `exceptional string (exceptional string α)`.

Example. Revisiting the safe division example, the code should look very similar to Impe, except for the more verbose type signature.

```
term divide-safely (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

To implement nondeterminism monadically, we have a couple different options. The option closest to Impe's implementation is to a version of the `mutability` monad that statefully keeps track of a *seed* that acts as a source of nondeterminism. The random seed can be represented as a list, and to produce a random value the head of the list is removed. The following implements this scheme for nondeterminism:

```
type seeded-nondeterministic (θ α : kind) := mutable (list θ) α.

term seeded-select (θ:Type) : seeded-nondeterministic θ θ
  := do
       { let l ← get
       ; set (tail l)
       ; lift (head l) }.

term seed (θ α : kind) : list θ → seeded-nondeterministic θ α → α
  := initialize.
```

Note that `seeded-nondeterministic` inherits its monad instance from `mutable`.

There is a more interesting possibility though. Suppose we would like to collect all the possible results of a nondeterministic computation. A useful structure for this is in fact the simple `list` data type, and we can use it directly as our monad of choice.

Listing 3.11: The nondeterminisic monad

```
type nondeterministic : kind → kind := 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) }.
```

Note that `map` is defined recursively, but is guaranteed to terminate because lists are constructed and thus are finite.

Example. Recall the `coin-flip` function presented in Impe. Using the `nondeterministic` monad we can implement the safe function in Mona:

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

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

The term `lucky` flips two coins and results in `true` only if both flips were "heads." The following reduction demonstrates the monadic computation of each of the four computational pathways:

```
do{ let a ← coin-flip; let b ← coin-flip; lift (a ∧ b) }
 ↠
coin-flip >>= (a ⇒ coin-flip >>= (b ⇒ lift (a ∧ b)))
 ↠
coin-flip >>= (a ⇒ coin-flip >>= (b ⇒ [a ∧ b]))
 ↠
coin-flip >>= (a ⇒ concat (map (b ⇒ [a ∧ b]) [true, false]))
 ↠
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

For external effects, the implicit context is an interface to some external implementation that handles the effect. In this way, it is impossible to provide a constructively-instantiated monad instance for external effects. So, external monadic effects are introduced in Mona via primitively-declared types that are primitively-instantiated as monads. This introduces potential danger because even though the external implementation may not behave as expected or even meet the requirements of `Monad`, the specifications and behavior are not expressible within Mona and thus Mona must assume a correct interface regardless.

3.4.2.1 I/O

We use I/O as a canonical example of an external effect. In Impe, the I/O effect was facilitated by an implicit reduction context $\mathcal{IO}$ that responded to commands of the form $\mathcal{IO}$(`input` •) or $\mathcal{IO}$(`output` *s*), where `s:string`. Here we declare a primitive type `io`, the monad for the I/O effect, that acts as an explicit version of $\mathcal{IO}$. And, we declare primitive terms `input` and `output` that provide the input and output capabilities of I/O and have the appropriate types making use of `io`.

Listing 3.12: Definitions for the monadic I/O effect

```
primitive type io : kind → kind.
primitive instance Monad io.

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

We still must include $\mathcal{IO}$ in the implicit reduction context for Mona, but its usage is now explicitly typed by `io` and so no longer hides whether or not a term uses the I/O effect. As the terms `input`, `output` and the terms defined by the `Monad io` instance are declared primitively, we must provide additional reduction rules to describe how they interact with $\mathcal{IO}$.

Table 3.2: Reduction in Mona: I/O

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

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

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

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

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

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

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

(The specification of $\mathcal{IO}$ is inherited from Impe.) Here, a term of the form $\{\!\{a\}\!\}$ indicates that the term was produced by a computation using the I/O effect. There is no way to reduce from a $\{\!\{a\}\!\}$ to a pure a since the results of using the I/O effect cannot be removed. Making use of this, it is common for programming languages using monadic effects, such as Haskell, to require a top-level term *main* of type `io unit` that is evaluated when the program file is run.

The following implements the familiar greeting example using the monadic I/O effect.

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

3.5 Considerations for Monadic Effects

Monadic effects offer a strategy for implementing effects that preserves many of the advantages of Func over Impe. Individual effect monads — instances of the `Monad` type-class — tag terms that use their effects. In so doing, the use of effects is considered explicitly in the writing of Mona programs, allowing better reasoning about effects than implicit, imperative effects do. Additionally, the monadic structure captured the idea of and specification for "effect" in terms of Mona code, rather than having to rely on an unenforced, external definition. Altogether, these features count well towards the safety of Mona's effects. Jus-

tifications like these have been forwarded by the likes of [8] and [11] in order to propose the use of monadic effects in industry programming languages. The most popular result of this movement is use of monadic effects as the core effect framework in the Haskell programming language.

However, these benefits do not come without costs. In [9], some drawbacks to monadic effects are detailed (relating to Haskell in particular, and along with some suggested extensions). The main problem is that monadic effects are difficult to use for real-world applications, which stems from two sources: managing monads is difficult for a programmer, and monadic effects do not compose.

In regards the first source: while the explicit typing of effects is useful for reasoning it also makes programs more complicated to write. In Impe, a programmer need only consider result types and computations were handled implicitly. In Mona, a programmer must consider both the result types and any monads being worked within. The `do` notation assists this, but there is still the complication of always needing to explicitly lift pure terms into trivial computations (via `lift`) and likewise for pure mappings (via `map`).

In regards to the second source: the problem of composing monadic effects is the problem of monads not playing well together. To compose monadic effects is to use different monadic effects in sequence. For example, suppose we want to write an integer-stateful computation `safely-divide-state` with parameter *i* `: integer` that divides *i* safely by the state and (only if the quotient is defined) then sets the new state to be the result of this. In Mona this function must be look something like the following:

```
term safely-divide-state (i:integer) : mutable integer unit
  := do
       { let j ← get
       ; let q := catch{ i ⇒ j } in safely-divide i j
       ; set q }.
```

But notice, however, that we would not have been able to write it with the `catch` statement on the outside of the `mutable`'s do block, like so:

```
term safely-divide-state (i:integer) : mutable integer unit
  := catch{ i ⇒ j } in
     do
       { let j ← get
       ; let q := safely-divide i j
       ; set q }.
```

This term is not well-formed because the type of `q` is `exceptional integer integer` but the next line's `set q` expects `q` to instead have type `integer`. The `catch` statement is intended to resolve this, but since it occurs outside of the mutable effect it cannot apply before the mutability effect.

The inability to compose monadic effects imposes a strict, hierarchical structure to using them. As demonstrated by `safely-divide-state`, the effects are forced to be done at different levels rather than sequenced in parallel (i.e. composed). Moving forward, an

inspiration is to implement effects in a way that maintains the advantages of Mona but also allows for composable effects. In the next chapter, Alge's algebraic effect handlers take a different approach to implementing effects, which achieves composability of effects and simpler types, but sacrifices some of the expressivity and type-safety of Mona.

Chapter 4

Algebraic Effects with Handlers

4.1 Breaking Down "Effect"

In chapter 2, we considered language Impe which implemented effects by introducing specific language features for each kind of effect. In chapter 3, we considered language Mona 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 Func that provides composable effects while also maintaining the useful features of Mona: type-safety, general class-like structure for new effects, internal effects.

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 external reduction contexts. This abstract strategy was to take something intrinsic about the nature of effects in general, and represent it explicitly in the 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 Impe: effects are performed by using specific primitive values, and are handled during reduction to affect the program state. In Mona: 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.

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

- **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).

In the following sections, Alge will introduce many new syntactical structures for describing algebraic effects, but the basic idea is all Func-terms of continuations. The algebraic effect strategy is two-fold: (1) performances build up a sequenced of nested continuations, each parametrized by their expected result, and (2) handlers, applied to a built-up continuation stack, handle each continuation in sequence by providing the performance's expected result and keeping track of a context relatively implicit to the performances themselves.

Additionally, algebraic effects need some way of particularizing effects by the performances allowed to be done by them, so that handlers can handle each performance case. The approach taken by [3] in designing the *Eff* programming language is to use the following structures.

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

A resource specifies the typed interface (collection of actions) to an effect, and the channels of that resource are particular instances of the effect (such as how there can be multiple terms of type `mutable` α for fixed α). Since a resource specifies an explicit collection of actions that act as atomic performances, handlers can account for any kind of performance using the resource by handling in terms of its actions.

The core idea of algebraic effects is to define an algebra on the performances that allows for the free composition of effects, and then to separately define algebraic effect handlers that can be applied hierarchically to performances. They were mainly introduced by [10], [12] and [2], and are inspired by a history of research into the actor model of computation, the π calculus [7], continuation-passing style, delimited control. The *Eff* programming language described by [3] is a variant of the OCaml programming language that implements algebraic effects with handlers in a way very similar to the language Alge introduced in the next section.

4.2 Language Alge

Language Alge implements algebraic effects with handlers in terms of resources, actions, channels, performances, and handlers.

4.2.1 Syntax

Table 4.1: Syntax for Alge

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

4.2.1.1 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 Func reduction rules for function applications do not apply to it. An action is a function that raises an *appeal* 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.

4.2.1.2 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 ρ`:Resource` and the terms $g_1 : \alpha_1 \nearrow \beta_1, \ldots, g_n : \alpha_n \nearrow \beta_n$.

The typing rule is as follows:

Table 4.2: Typing in Alge: Resource

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

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`.

4.2.1.3 Channel

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

```
channel c : ρ.
```

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

Table 4.3: Typing in Alge: Channel

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

For example, consider the following channel declarations:

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

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

[1] 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 Alge.

4.2.1.4 Performance

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

```
c#g a
```

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

Table 4.4: Typing in Alge: Performance

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

For example, consider the following term:

```
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.

4.2.1.5 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 4.2.1.6) 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 $g : \chi \nrightarrow \upsilon$ has the form `#g x k ⇒ b`, where `x`,`k` are term parameters and `b` is a term. Here, `x` is an input parameter for `g`, `k` is a continuation parametrized by the result of performing `g`, and `b` is a term that encodes the result of performing `g` given input `x` and continuation `k`. More specifically, `k` encodes the rest of the computation to carry out after `g` is performed, with a parameter of type χ which appears everywhere the result of performing `g` normally appeared (even if it does not appear anywhere).

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

Firstly, a handler's *relatively-pure value clause*, or just **value clause**, encodes a lifting of relatively-pure values to be of the appropriate type reflecting the effect handling (note that this can be the trivial lift, `a ⇒ a`). Such a value clause that handles relatively-pure values has the form $a \Rightarrow b$, 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 (see Appendix: Preludes, Func) 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 α and has result type β is $\alpha \searrow \beta$. The " $\searrow$ " is the the over-horizontal reflection of the action type's operator, indicating that a handler *resolves* (or, *lowers*) the raised 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 b_f ⇒ c_f }
```

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

Table 4.5: Typing in Alge: Handler

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

For example, consider the following term:

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

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

4.2.1.6 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 4.2.2.

The typing rule is as follows:

Table 4.6: Typing in Alge: Handling

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

For example, consider the following term (section 4.2.1.7 describes the `do` syntax):

```
term experiment : boolean
  := let b := random1#gen-boolean • in
     let p := random1#gen-probability • in
     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`.

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
```

4.2.1.7 Sequencing

The sequencing effect can be constructed in Alge, since the usual reduction rules inherited from Func will first reduce the first argument of `sequence` and then reduce the second argument. In this way, the performances of the first argument are performed first and the second arguments' are performed second. Additionally, we need not introduce a special notation for binding since, unlike in Mona, the results of an α-performance has the same type as a pure α-term. Instead of dealing with effect-tagged types as the performance level, Alge allows dealing with types to be raised out to the handling level.

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)`

4.2.2 Reduction

Alge introduces many new syntactical structures for which reduction rules must be provided. In chapter 3, we discussed how relying on reduction rules contributed to danger, since

reduction rules are not expressible within the programming language itself. However, the important reasoning for this was that the reduction rules relied on implicit contexts in order to perform effects in a way not internally-expressible. In contrast, Alge's new syntactical structures provide an internal way of defining direct interactions with the reduction context, rather than using the reduction context as purely an interface. So, even though Alge requires quite a few more reduction rules, the behavior of algebraic effects with handlers is still able to be directly reasoned about (with some amount of locality) within Alge code itself.[2]

Reduction contexts. The total reduction context Δ is made up of two sub-contexts: the handlers context $\mathcal{H}$, and the performances context $\mathcal{P}$. 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 h1) with h2`, 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{ a1 ; a2 ; a3 }` would yield the performance context $\mathcal{P} = [a_3, a_2, a_1]$.

Relative Values. Recall from section 1.2.3 that a term is a value if no reduction rules can simplify it. Note though that terms in Impe must rely on relatively implicit contexts for reduction — in particular, $\mathcal{H}$. For example: `random1#gen-boolean •` is a value if it appears at the top level, but the same term is not a value if it appears somewhere within the appropriate handling structure, such as `random1#gen-boolean • with not-so-randomly random1`. So, the proposition that a term v is a value, written value v, must be extended to include the reduction context. Thus we introduce the new form "value v relative to $\mathcal{H}$" to abbreviate "v is a value relative to handler context $\mathcal{H}$." The proposition "value v relative to $\mathcal{H}$" is true if value v and there is no handler h among the h such that either of the following is true: v is an action and h has a matching action clause, or h has a `value` clause.

Simplification. When reducing the argument in a function application, any handlers used in this sub-term should not be propagated to the parent level. For example, when reducing the term `(a1 (a2 with h2)) with h1`, the handler h_2 clearly shouldn't be used to handle a_2. On the other hand, when reducing a term `a (r#e x) with h`, the performance `r#e x` *should* be raised to the parent level in of `a (r#e x)`, since the hole introduced by `Perform` should be put in place of the argument of a. Altogether, simplification of the argument in function application should add performances to $\mathcal{P}$, but should not add handlers to $\mathcal{H}$.

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 $(r, g, v, @)$ to the head of the performances context $\mathcal{P}$, where $@$ is a fresh term

[2]The language *Eff* from [3] uses a different reduction scheme that does not require reduction contexts — it passes around the `a with h` handler application to sub-terms instead that a reduction context. This is an alternative and attractive approach, though with slightly less readability in my opinion, and further demonstrates how algebraic effects are more analyzable than naively-imperative effects.

name that stands in place for the result of the performance.[3] This indicates how $(r, g, v, @)$ is the new inner-most performance, first in priority to get considered for handling. The result of this reduction rule is just @, which will be filled in by whatever the handled result of $(r, g, v, @)$ 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.

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.

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 $(r, g, v, @)$. 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 `@ ⇒ w`, since @ stands in place of the result of the performance and w is a term referencing @ to describe the rest of the computation. So altogether the result of HANDLE-PERFORMANCE is b with v passed as x and `@ ⇒ w` passed as k, written as the application `(x k ⇒ b) v (@ ⇒ w)`.

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

Observe that HANDLE-VALUE has a higher priority than HANDLE-PERFORMANCE. The result of this is that, once all the performances have been dictated by PERFORM, rule HANDLE-VALUE will apply once, and then the performances will be subsequently handled.

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

[3]Note that the name @ indicates that it is a fresh variable name, named so because it provides a sort of "hole" in the term it is abstracted from.

Handler modifiers. The technique used by the reduction rules for Alge in order to keep track of which of the steps of reduction (performance, value, then final) a handler is in is to modify the handler so that only the correct reduction rule applies to the handler's structure at each step. The unvalue modifier removes the **value** clause, the unfinal modifier removes the **final** clause, and the monoperforms modifier removes both the **value** and **final** clauses from a given handler.

Table 4.7: Reduction in Alge

$$\textsc{Simplify}\quad \frac{\mathsf{monoperforms}(h) :: \mathcal{H}\ ;\ P \parallel a \twoheadrightarrow \mathsf{monoperforms}(h) :: \mathcal{H}\ ;\ P' \parallel a'}{h :: \mathcal{H}\ ;\ P \parallel a\ b \twoheadrightarrow h :: \mathcal{H}\ ;\ P' \parallel a'\ b}$$

$$\textsc{Simplify}\quad \frac{\text{value } v \text{ relative to } \mathcal{H} \qquad \mathsf{monoperforms}(h) :: \mathcal{H}\ ;\ P \parallel b \twoheadrightarrow \mathsf{monoperforms}(h) :: \mathcal{H}'\ ;\ P' \parallel b'}{h :: \mathcal{H}\ ;\ P \parallel v\ b \twoheadrightarrow h :: \mathcal{H}'\ ;\ P' \parallel v\ b'}$$

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

$$\textsc{Handle}\quad \mathcal{H}\ ;\ P \parallel a\ \mathbf{with}\ h \twoheadrightarrow h :: \mathcal{H}\ ;\ P \parallel a$$

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

$$\textsc{Handle-Final}\quad \frac{\text{value } v \text{ relative to } \mathsf{unfinal}(h) \qquad h := \mathbf{handler}\ \{\ldots\ \mathbf{final}\ b \Rightarrow c\ ;\ \ldots\}\ r}{\mathsf{unvalue}(h) :: \mathcal{H}\ ;\ P \parallel v \twoheadrightarrow \mathcal{H}\ ;\ P \parallel (b \Rightarrow c)\ v}$$

$$\textsc{Handle-Value}\quad \frac{\text{value } v \text{ relative to } h :: \mathcal{H} \qquad h := \mathbf{handler}\ \{\ldots\ \mathbf{value}\ a \Rightarrow b\ ;\ \ldots\}\ r}{h :: \mathcal{H}\ ;\ P \parallel v \twoheadrightarrow \mathsf{unvalue}(h) :: \mathcal{H}\ ;\ P \parallel (a \Rightarrow b)\ v}$$

$$\textsc{Handle-Performance}\quad \frac{\text{value } w \text{ relative to } h :: \mathcal{H} \qquad h := \mathbf{handler}\ \{\ldots\ \#g\ x\ k \Rightarrow b\ ;\ \ldots\}\ r}{\mathsf{unvalue}(h) :: \mathcal{H}\ ;\ (r\#g\ v,\ @) :: P \parallel w \twoheadrightarrow \mathsf{unvalue}(h) :: \mathcal{H}\ ;\ P \parallel (x\ k \Rightarrow b)\ v\ @(\ \Rightarrow w)}$$

4.2.3 Algebraic Effects with Handlers

4.2.3.1 Mutability

The mutability effect can be defined in Alge as a resource as follows, providing its two signature capabilities, `get` and `set`, here as the resource's actions.

Listing 4.4: Resource for mutability

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

The mutable value is handled in `initialize` using another layer of continuation that keeps track of the state. The current continuation `k` has two parameters: the current mutable value and the result.

Listing 4.5: 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.

Example. 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
  := let i := counter#get • in
     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:

```
h := initialize s0 counter.

[h] ; [] ‖ increment •
(Definition)
[h] ; [] ‖ let i := counter#get • in 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, •)
```

In this reduction, the handling of `increment •` yields a continuation `s ⇒ (s + 1, •)` which encodes the transforming of an initial integer state `s` to a affected integer state `s + 1` and a resulting value •. This state modification and result value are appropriately expected from the construction of the `increment` function. Abbreviate the reduction done by Handle-Performance x2 as Handle-Increment. Now, consider the following experiment using `increment` to be handled.

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

4.2.3.2 Exception

The exception effect can be defined in Alge as a resource that provides one action for throwing exceptions. A "valid" action is not needed since pure terms are by default considered valid (which is possible since algebraic effect handlers postpone the type-validation to handling rather than requiring it when performing).

Listing 4.6: Resource for exception

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

The resource `Exceptional` is parametrized by its result type.

Example. The resource for the exception effect can be instantiated as particular exception channels, where each channel corresponds to a particular sort of exception. For example, we can write the `divide-safely` in Alge as the following, where we first declare a new channel `division-by-0` representing the exception of dividing by `0`.

```
channel division-by-0 : Exceptional integer.

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

Handler for exceptions. We can write the following canonical `Exceptional`-handler `trying` with a similar structure to how the `exceptional` monad was defined in Mona. If a `#throw` e action is performed, then the continuation k is ignored and `left e` is propagated as the result. The `value` clause treats values as valid and so uses `right`, the α-constructor of $\varepsilon \oplus \alpha$.

Listing 4.7: Handler of exceptions as optionals

```
term trying (α:Type) : Exceptional ε → α ↘ ε ⊕ α
  := handler
       { #throw e k ⇒ left e
       ; value  a   ⇒ right a
       ; final  x   ⇒ x }.
```

We can handle a performance of `divide-safely` by using this handler. If the performance throws a `division-by-0` exception, then `trying division-by-0` will handle it to `none`. if the performance throws no exceptions and results in a, then `trying division-by-0` will handle it to `right` a.

```
term h := trying 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 •, @1)] ‖ @1
↠ (Handle-Value)
[unvalue(h)] ; [(division-by-0#throw •, @1)] ‖ some @1
↠ (Handle-Performance)
[unvalue(h)] ; [] ‖ (x k ⇒ none) • (@1 ⇒ some @1)
↠ (Simplify)
[unvalue(h)] ; [] ‖ none
↠ (Handle-Final)
[] ; [] ‖ none
```

4.2.3.3 Nondeterminism

We shall use the example of coin-flipping to demonstrate algebraic effect-handled nondeterminism. A nondeterministic coin-flipping effect requires one action: flipping a coin to get either "heads" or "tails." A handler for this effect in some way fills in the result of each coin flip.

Resource. The resource `Coin` is the resource for the nondeterministic coin-flipping effect. It provides one action, `flip`, which results in a `boolean` value where `true` corresponds to "heads" and `false` corresponds to "tails".

Listing 4.8: Resource for nondeterministic coin-flipping

```
resource Coin { flip : unit ↗ boolean }.
```

Example An experiment that counts the number of heads resulting from two `coin#flip`'s. The following familiar experiment `lucky` flips two coins and returns true only when both were heads. In order to use the coin-flipping effect, we first declare a new channel `coin` of the `Coin` resource.

```
channel coin : Coin.

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

Uniquely for algebraic-effect-handling, the term `lucky` is inert on its own — a handler must be provided to handle the effects `lucky` encodes. Since the performances are abstracted from the handling in this way, we can define many different ways of handling the same effect. The following paragraphs demonstrate.

Handler for all possibilities. This handler accumulates all possible results into a list.

Listing 4.9: 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 }.
```

We can use `either-heads-or-tails` to handle `lucky`, yielding a list of all the possible results of `lucky`.

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

[] ; [] ‖ lucky with h
↠ (Handle)
[h] ; [] ‖ (coin#flip •) ∧ (coin#flip •)
↠ (Perform ×2)
[h] ; [(coin#flip •, @2), (coin#flip •, @1)] ‖ @2 ∧ @1
↠ (Handle-Value)
[h] ; [(coin#flip •, @2), (coin#flip •, @1)] ‖ [@2 ∧ @1]
↠ (Handle-Performance ×2)
[h] ; [] ‖ [true ∧ true, false ∧ true, false ∧ true, false ∧ false]
↠ (Simplify)
[h] ; [] ‖ [true, false, false, false]
↠ (Handle-Final)
[] ; [] ‖ [true, false, false, false]
```

Handler for fixed possibility. This handler determines each coin flip to yield heads.

Listing 4.10: Handler for just heads

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

We can use `just-heads` to handle `lucky`, yielding the result of `lucky` if each coin-flip was a "heads".

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

[] ; [] ‖ experiment with just-heads
↠ (Handle)
[h] ; [] ‖ (coin#flip •) ∧ (coin#flip •)
↠ (Perform ×2)
[h] ; [(coin#flip •, @2), (coin#flip •, @1)] ‖ @2 ∧ @1
↠ (Handle-Value)
[unvalue(h)] ; [(coin#flip •, @2), (coin#flip •, @1)] ‖ @2 ∧ @1
↠ (Handle-Performance ×2)
[unvalue(h)] ; [] ‖ true ∧ true
↠ (Simplify)
[unvalue(h)] ; [] ‖ true
↠ (Handle-Final)
[] ; [] ‖ true
```

Handler for alternating possibilities. This handler alternates the results of flips between heads and tails.

Listing 4.11: 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 }.
```

We can use `alternating-between-heads-and-tails` to handle `lucky`, yielding the result of `lucky` if the first coin-flip was "heads" and the second was "tails".

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

[] ; [] ‖ experiment with h
↠ (Handle)
[h] ; [] ‖ (coin#flip •) ∧ (coin#flip •)
↠ (Perform ×2)
[h] ; [(coin#flip •, @2), (coin#flip •, @1)] ‖ @2 ∧ @1
↠ (Handle-Value)
[unvalue(h)] ; [(coin#flip •, @2), (coin#flip •, @1)] ‖ (b1 ⇒ @2 ∧ @1)
↠ (Handle-Performance)
[unvalue(h)] ; [(coin#flip •, @1)]
  ‖ (_ k ⇒ (b2 ⇒ k b2 (not b2))) • (@2 b1 ⇒ @2 ∧ @1)
↠ (Simplify)
[unvalue(h)] ; [(coin#flip •, @1)] ‖ b2 ⇒ b2 ∧ @1
↠ (Handle-Performance)
[unvalue(h)] ; []
  ‖ (_ k ⇒ b3 ⇒ k b3 (not b3)) • (@1 b2 ⇒ b2 ∧ @1)
↠ (Simplify)
[unvalue(h)] ; [] ‖ b3 ⇒ (not b3) ∧ b3
↠ (Handle-Final)
[] ; [] ‖ (f ⇒ f true) (b3 ⇒ (not b3) ∧ b3)
↠ (Simplify)
[] ; [] ‖ false ∧ true
↠ (Simplify)
[] ; [] ‖ false
```

4.2.3.4 I/O

The I/O effect presents here similarly to Impe's implementation in contrast to Mona's implementation. This is because, important unlike in Mona, I/O-performances that results in α is treated just like a pure α. Differently from both Impe and Mona, however, here we are able to present both an external handler and an internal handler for the I/O effect.

Resource. As usual, the I/O effect has two actions: `input` for receiving input, and `output` for sending output.

Listing 4.12: Resource for I/O

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

We provide a standard `IO`-channel to represent a generic I/O peripheral (e.g. a computer terminal). In many real-world programming languages this is referred to as *standard I/O* or just *stdio*.

Listing 4.13: Channel for I/O

```
channel io : IO.
```

Example The usual `greetings` example can be written in a very similar way here as it was in Impe, with just two differences. The first is that `greetings` does not require a `unit`-parameter, which was needed in Impe in order to postpone the evaluation (and thus I/O-effect) until the unit parameter was provided. Here the I/O effect is not performed until a handler is applied and so this is not needed. The second difference is that `input` and `output` are now proceeded by the `io` channel since performances are required to operate on a specific channel, rather than on an implicitly global and unitary $\mathcal{IO}$.

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

Handlerlessness of external I/O. In the same way that the `io` monad from Mona would not allow the result of a computation to be extracted from the `io` wrapper we should likewise not provide a handler of type `IO → α ↘ α`. We *could* provide a dangerous handler of the type `IO → α ↘ α`, but this would counter our purposes of moving on from Impe— allowing the "free" handling of `IO`-performances effectively removes any guarantee that I/O effects are explicitly represented in Alge.[4] Instead, it is assumed that `IO`-performances can only be handled at the top level when the program is executed by some unique handler provided at that time of type `IO → α ↘ unit`. This "main" handler for I/O programs would have an imaginary body looking something like the following:

Listing 4.14: Imaginary body for `external`

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

[4]Most real-world languages, even the most effectually-purist such as Haskell, provide dangerous handler-like capabilities such as this for I/O and other effects.

The $\mathcal{IO}$ used here inherits the same specification as given for Impe.

Listing 4.15: Handle greetings with standard I/O

```
h := external io.

[] ; [] ‖ greetings with h
↠ (Definition greetings)
[] ; [] ‖ io#output ("Hello, " ++ io#input •) with h
↠ (Handle)
[h] ; [] ‖ io#output ("Hello, " ++ io#input •)
↠ (Perform)
[h] ; [(io#input •, @1)] ‖ io#output ("Hello, " ++ @1)
↠ (Simplify)
[h] ; [(io#input •, @1)] ‖ io#output ("Hello, " ++ @1)
↠ (Perform)
[h] ; [(io#output ("Hello, " ++ @1), @2), (io#input •, @1)] ‖ @2
↠ (Handle-Value)
[unvalue(h)] ; [(io#output ("Hello, " ++ @1), @2), (io#input •, @1)]
  ‖ @2
↠ (Handle-Performance)
[unvalue(h)] ; [(io#input •, @1)]
  ‖ (x k ⇒ k 𝒪(output x)) ("Hello, " ++ @1) (@2 ⇒ @2)
↠ (Simplify)
[unvalue(h)] ; [(io#input •, @1)] ‖ 𝒪(output ("Hello, " ++ @1))
↠ (Handle-Performance)
[unvalue(h)] ; []
  ‖ (x k ⇒ k 𝒪(input x)) • (@1 ⇒ 𝒪(output ("Hello, " ++ @1)))
↠ (Simplify)
[unvalue(h)] ; [] ‖ 𝒪(output ("Hello, " ++ 𝒪(input •)))
↠ (Simplify)
[unvalue(h)] ; [] ‖ 𝒪₂(output ("Hello, " ++ 𝒪₁(input •)))
↠ (Simplify)
[unvalue(h)] ; [] ‖ •
```

Note that the usage of the interface I/O interface $\mathcal{IO}$ is subscripted by the order of use. So, in the last Simplify step, the term $\mathcal{IO}_1$`(input •)` is evaluated first via $\mathcal{IO}$, and then the outer $\mathcal{IO}_2$`(output ("Hello, " ++ `$\mathcal{IO}_1$`(input •)))` is evaluated via $\mathcal{IO}$.

Handler for internal I/O The previous section made use of a primitive, implicit handler for the I/O which appealed to a language-external interface $\mathcal{IO}$. 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 an internal handler for I/O.

The I/O effect requires a stream of inputs and a store for outputs. We can represent the 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 description:

Listing 4.16: Handler for pure I/O

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

The implementation here derives its complexity from the two continuation layers — the parameters `is` and `os` to the performance and `value` branches — added to keep track of the current stream of inputs and store of outputs. The strategy is exactly the one used by the handler `initialize` for mutability, but with two layers rather than just one.

The following demonstrates how `internal` handles `greetings` to produce a stream of outputs matching what would be expected to print to the console when using a main external handler.

Listing 4.17: Handle greetings with internal I/O

```
h := internal ["Henry", "Blanchette"] io.

[] ; [] ‖ greetings with h
↠ (Handle)
[h] ; [] ‖ greetings
↠ (Definition)
[h] ; [] ‖ io#output ("Hello, " ++ (io#input •))
↠ (Perform)
[h] ; [(io#input •, @1)] ‖ io#output ("Hello, " ++ @1)
↠ (Perform)
[h] ; [(io#output ("Hello, " ++ @1), @2), (io#input •, @1)] ‖ @2
↠ (Handle-Value)
[unvalue(h)] ; [(io#output ("Hello, " ++ @1), @2), (io#input •, @1)]
  ‖ is os ⇒ (@2, os)
↠ (Handle-Performance)
[unvalue(h)] ; [(io#input •, @1)]
  ‖ (x k is os ⇒ k • is (os ◇ [x]))
      ("Hello, " ++ @1) (@2 is os ⇒ (@2, os))
↠ (Simplify)
[unvalue(h)] ; [(io#input •, @1)]
  ‖ is os ⇒ (•, os ◇ ["Hello, " ++ @1])
↠ (Handle-Performance)
[unvalue(h)] ; []
  ‖ (_ k is os ⇒ k (head is) (tail is) os)
      • (@1 is os ⇒ (•, os ◇ ["Hello, " ++ @1]))
↠ (Simplify)
[unvalue(h)] ; [] ‖ is os ⇒ (•, os ◇ ["Hello, " ++ head is])
↠ (Handle-Final)
[] ; []
  ‖ (is os ⇒ (•, os ◇ ["Hello, " ++ head is]))
      ["Henry", "Blanchette"] []
↠ (Simplify)
[] ; [] ‖ (•, ["Hello, Henry"])
```

4.3 Considerations for Algebraic Effects with Handlers

Algebraic effects with handlers offer a strategy for implementing effects that preserves some of the advantages of Mona over Impe, while also allowing effects to be used much more easily than in Mona. Recall that the core idea of algebraic effects with handlers is the division of effects into performances and handlers. This framework allowed for Alge to allow the free

composition of performances, which was a problem with Mona mentioned in section 3.5. Computations are not tagged by an wrapper effect type for performance, and so their result terms can be used among pure terms as if they were pure — the blissful ignorance of Impe. That is composability of effects. Handlers still remain strictly hierarchical.

While allowing composable performances, Alge manages to not suffer as much from the un-analyzability of Impe's effects; in Alge effects must be explicitly handled, where the typing scheme for handlers salvages some type-safety. If a handler is applied to a term, it is inferable that all performances of the handler's effect have been handled.

However, there are still some drawbacks to algebraic effects with handlers. Though composability of effects is achieved without entirely losing type-safety, there is still some type-safety lost. Code cannot be analyzed as locally as Mona code, since performances rely on (top-level explicit) reduction contexts to dictate reduction. It is an interesting new ability for a performance `output "hello"` to be possibly handled by many different handlers, but it also implies that it is hard to determine what `output "hello"` *is* on its own. It is a syntactically new structure beyond Func's vocabulary. There is an analogy that captures a Func-like intuition for these new structures:

functions are to arguments (in Func) as handlers are to performances (in Alge)

In other words, handlers are like a special kind of function term that specifically accepts performances (and trivially accepts pure terms) as input. Might there be a way of encapsulating this structure within an extension of Func in a similar way to how `Monad` captured an idea of effects? Such an encapsulation would greatly shrink the complexity of Alge's syntactic and reduction definitions.

Another significant drawback to algebraic effects with handlers is that, though applying a handler to a term guarantees that performances of its effect have been handled in the term, it is still indeterminable from the type of the term whether or not it uses certain effects in the first place. This problem also arose in Impe but was avoided in Mona. Consider the situation where a library provides a black-box function `do-something` of which it is undeterminable if it uses certain effects, then the function is dangerous. Some language-internal feature (read: types) specifying usage of effects is what we desire.

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

In section 3.5, we reflected on this problem as the problem of *composing monads* in Mona. For Mona, 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 Alge 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 Mona 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

[1] There is another monadic structure called *monad transformers* that also facilitate this feature. However, they are not as general or easy to use as freer monads.

implement a variant of algebraic effect handlers with Func using monads. The strategy is to define a generalized monad that is parametrized by an effect type as well as a result type.

We will construct a type `freer : (kind → kind) → kind → kind` which lifts particular effects of type `kind → kind` 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 : kind → kind) (α : kind) : 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 Frer.

5.2 Freer Monad

Freer monads are majorly described as a basis for a generalized effect framework in [6]. However that work 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 : kind → kind` 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 : kind → kind`, to define a type `freer : (kind → kind) → kind → kind` 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 : kind → kind) (α : kind) : kind
  := pure   : α → freer υ α
   | impure : (χ:kind) ⇒ υ χ → (χ → freer υ α) → freer υ α.
```

The constructor `pure` clearly corresponds to the lifting monad capability. It is named so to reflect the lifting of a pure `α` to the impure type `freer υ α`[4] The motivation for constructor `impure` is less obvious. It's type signature appears as a sort of mixing between the binding

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

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

[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: Preludes, Mona) 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.

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

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

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

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

Listing 5.2: Monad instance of `freer`

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

Indeed this results in a monad instance for any υ.

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

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

5.3 Language Frer

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

```
type mutable-base (σ α : kind) : kind := σ → σ × α.
type mutable      (σ α : kind) : kind := 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-lift` like so:

```
term get-base (σ:kind) : mutable-base σ σ := s ⇒ (s, s).
term get      (σ:kind) : mutable      σ σ := freer-lift (get-base •).

term set-base (σ α : kind) (s:σ) : mutable-base σ unit := _ ⇒ (s, •).
term set      (σ α : kind) (s:σ) : mutable      σ unit := freer-lift
                                                          (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-lift «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:kind) ⇒ mutable σ
 lifts (α:kind) ⇒ σ → σ × α
  { get       := 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 : kind) (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 multiple interesting handlers.

Now we can write the freer-monadic version of the `sort-two` example of mutability.

```
type state := { x:integer; y:integer }.

term get-x : mutable state integer := get >>= (lift ∘ x).
term set-x : mutable state integer := get >>= (lift ∘ y).

term modify (f : σ → σ) : mutable σ α → mutable σ α
  := get >>= (set ∘ f).

term set-x : integer → mutable state integer := modify map-x ∘ constant.
term set-y : integer → mutable state integer := modify map-y ∘ constant.

term sort-two : mutable state unit
  := do
      { let vx ← get-x
      ; let vy ← get-y
      ; if vx > vy
          then set-x y >> set-y x
          else lift • }.
```

The function `constant (α β : kind) : α → β → α` is given by `constant a b := a`. This implementation of the mutability effect behaves just the same as the normal monadic effect.

```
initialize (1, 2) sort-two
↠ ⋯ ↠
initialize (1, 2, true)
  ( impure (s ⇒ (s, s)) (s ⇒
      impure (s ⇒ (s, s)) (s ⇒
        if (x s) > (y s)
          then
            ( impure (s ⇒ (s, s)) (s ⇒
                impure (s _ ⇒ (s, •)) pure ∘ map-x ∘ constant (y s) s)
            >>impure (s ⇒ (s, s)) (s ⇒
                impure (s _ ⇒ (s, •)) pure ∘ map-y ∘ constant (x s) s) )
          else pure •)) )
↠ ⋯ ↠
((2, 1), •)
```

There are a lot of complicated reduction steps omitted here, because it is hard to follow and is better understood by the abstract method of simulating algebraic effect handling with monads.

5.3.0.1 Exception

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

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

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

The handler `trying` 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 trying (ε α : kind) (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 : kind) (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 Frer 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-lift (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-lift (left 4))
↠
catching (i ⇒ 0) (impure (left 4) pure)
↠
```

```
(i ⇒ 0) 4
↠
0
```

5.3.0.2 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 : kind) (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 }
```

(For the construction of `map` in the `Monad` instance for `list`, see Appendix: Preludes, Func). Recall the coin-flipping experiments from chapters 3 and 4. We can represent the sample list of coin-flipping as `[true, false]`, as in 3. The experiment is written in Frer as follows:

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

term lucky : 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 `lucky`.

```
all-possibilities lucky
↠ (Definition of lucky)
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]
```

5.3.0.3 I/O

For the I/O effect, Alge 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 : kind → kind)
  { input-class  : unit → ιo string
  ; output-class : string → ιo unit }.

effect (ιo : kind → kind) {IO ιo} ⇒ io ιo
 lifts ιo
  { input  _ := input-class •
  ; output s := output-class s }.
```

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

```
primitive type external : kind → kind.

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.
```

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

```
%
type internal (α:kind) : kind
  := 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 : kind → kind) {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.
```

5.4 Stacked-Freer Monad

The goal in adopting freer monads as a framework for effects was to vary monadic effects in a way that allows for composable effects. However, the `freer` type given in this chapter only appears to facilitate the same organization of effects as Mona— the freer-monadic effects given in this chapter don't compose as given so far. What is missing is a way of taking a term m `: freer` υ α and a term fn `: ` α `→ freer` ζ β and binding them (notice crucially that the base types υ and ζ do not necessarily match). The given `Monad` instance for `freer` is parametrized by a single base type, and is not compatible with mutliple base types at once. What is missing from Frer is something corresponding to the Alge's reduction context $\mathcal{H}$ containing a stack of handlers.

A solution to this is *stacked freer monads*, which are implemented by the Polysemy[5] package for Haskell. A **stacked-freer monad** is a freer monad that has base type `Σ : Base-Stack`, where `Base-Stack` α is the kind of stack of base types parametrized by a shared result type α. The base stack `Σ` in m `: freer Σ` α contains the base types of each effect used in m. In order to monadically bind the result of m to the parameter of the continuation fm `: ` α `→ freer Σ′` β, the stacks are concatenated into a result of type `freer (Σ ⬡ Σ′)` β (where the `⬡` operator concatenates `Base-Stack`-types). Thus the composing of effects is achieved! There are many details missing here, but unfortunately a large toolbox of type-level operations are required in order to formally express them — they are beyond the scope of this work.

Finally, handlers with types of the form `freer (υ ∴ Σ)` α `→ freer Σ` α can be constructed in order to handle effects on the base stack, popping them off of the base stack in the same way that handlers were popped of of $\mathcal{H}$ when all their performances were handled. In Mona and Frer we have already demonstrated what these handlers look like e.g. `initialize`, `trying`, `catching`, `all-possibilities`, `internal`, `external`. These handlers follow Alge's style of providing a heirarchy, since only one handler can be applied at a time.

[5] https://hackage.haskell.org/package/polysemy

Once the base stack is handled down to the empty stack, a handler of type `freer () α → α` can reduce the trivial computation (uses no effects) to a proper pure term.

5.5 Considerations for Freer-Monadic Effects

The freer-monadic framework for defining effects, extended by stacked-freer monads, provides meets almost all of the goals proposed throughout this work. Freer monads take full advantage of the type-safety of Func and Mona while also providing composable effects in an algebraic-effects-with-handlers style. Recall the `get-username` example from the beginning of this chapter. Without filling in all the details, we could write it freer-monadically as something the following.

```
type store : kind { username:string }.

term get-username (_:unit)
  : freer (io, mutable store, exceptional string) string
  := do
       { let name ← stacked input
       ; stacked (set-username name)
       ; if name == "Henry"
           then stacked (throw "invalid name")
           else lift name }.
```

The function `stacked` takes a freer-monadic performance and lifts it to the appropriate stacked-freer-monadic performance. The effects are interleaved, and the composition of using multiple effects is reflected in the type of `get-username`. Handlers need to be provided to handle each of the effects off of the `Base-Stack` in order to perform the effects.

A major drawback to freer-monadic effects is that the type-system it requires is very complex and not very programmer-friendly. It seems like a lot of work on the programmer's part to have to appeal to a highly abstract structure like freer monads just in order to write a Hello World program. However, most of the complexity of freer-monadic effects, as demonstrated in by example above, can be mostly hidden from the programmer in most typical use-cases.

Chapter 6

Appendix: Conventions

6.1 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.`"

6.2 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. E.g. "the usual."
- **italic font**: emphasis in narrative language, especially of new and important words. E.g. "*emphasize this*."

- **bold font**: indicating the definition of new terms keywords in narrative language. E.g. "**important definition**."
- **small-caps font**: names in logical meta-language. E.g. "The-Golden-Rule."
- **sans-serif font**: mathematical meta-language. E.g. "$f(x) = (f \circ f)(x)$."
- **monospace font**: programming language. E.g. "`this is some code`."

6.3 Names

Naming conventions in programs:

- **terms**: lower-case english word/phrase in kabob-case. E.g. `this-is-a-term`, `this-is-not-a-term`, `map`.
- **term variables**: lower-case english letter. E.g. `a` `b` `c`.
- **types**: lower-case english word/phrase. E.g. `list`, `optional`, `unit`.
- **type-classes**: capitalized english word/phrase. E.g. `Monad`, `Animal`, `Functor`.
- **type variables**: lower-case greek letter. E.g. `α`, `β`, `ϕ`, `υ`.
- **type variables of higher order**: capital english letter. E.g `M`, `A`, `L`.

Chapter 7

Appendix: Language Definitions

7.1 Func

7.1.1 Syntax

Table 7.1: Syntax for Func

metavariable	generator	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*»	`kind`	atom
	`kind →` «*kind*»	arrow
«*type*»	«*type-name*»	atom
	`(`«*type-param*»`:`«*kind*»`) ⇒` «*kind*»	function
	«*kind*» «*kind*»	application
«*term*»	«*term-name*»	atom
	`(`«*term-param*»`:`«*type*»`) ⇒` «*term*»	function
	«*term*» «*term*»	application

7.1.2 Notation

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 Func, but they can completely reduce to the core set of syntactical structures as defined by table 7.1. 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.

Declarations A Func program is a sequence of declarations, where declarations can only be written at this top level. Declarations don't *do* any computation but merely serve as a programmatic skeleton. In interpreting the steps of computing Func with programs — syntax-checking, type-checking, evaluating — declarations take effect right before type-checking. In this way, declarations can add judgements to the type-judgement context. They do so statically i.e. the order declarations are written in a program not matter.

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

Listing 7.1: Notation for multiple parameters

$$(\text{«}\textit{term-param}\text{»}_1 \texttt{ : } \text{«}\textit{kind}\text{»}_1) \cdots (\text{«}\textit{term-param}\text{»}_n \texttt{ : } \text{«}\textit{kind}\text{»}_n) \Rightarrow \text{«}\textit{term}\text{»}_*$$
$$::=$$
$$(\text{«}\textit{term-param}\text{»}_1 \texttt{ : } \text{«}\textit{kind}\text{»}_1) \Rightarrow \cdots \Rightarrow (\text{«}\textit{term-param}\text{»}_n \texttt{ : } \text{«}\textit{kind}\text{»}_n) \Rightarrow \text{«}\textit{term}\text{»}_*$$

$$(\text{«}\textit{type-param}\text{»}_1 \texttt{ : } \text{«}\textit{kind}\text{»}_1) \cdots (\text{«}\textit{type-param}\text{»}_n \texttt{ : } \text{«}\textit{kind}\text{»}_n) \Rightarrow \text{«}\textit{kind}\text{»}_*$$
$$::=$$
$$(\text{«}\textit{type-param}\text{»}_1 \texttt{ : } \text{«}\textit{kind}\text{»}_1) \Rightarrow \cdots \Rightarrow (\text{«}\textit{type-param}\text{»}_n \texttt{ : } \text{«}\textit{kind}\text{»}_n) \Rightarrow \text{«}\textit{kind}\text{»}_*$$

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 7.2: 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 7.3: Notations for multiple shared-type and shared-kind parameters

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

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

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

Listing 7.4: 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.

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`.

7.1.2.1 Primitives

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 Func. They are the arrow type, sum type, and product type.

Arrow Type The first notable primitive type is the *arrow type*, where `arrow α β` is the type of terms that are functions with input type α and output type β. All we need is the following declaration, since functions are a syntactic structure already introduced by table 7.1.

Listing 7.5: Primitive for the arrow type

```
primitive type arrow : kind → kind → kind.
```

Notation for infixed type arrow. This notation is right-associative. For example, the types `α → β → γ` and `α → (β → γ)` are equal.

Listing 7.6: Notation for infixed type arrow

```
«kind»₁ → «kind»₂    ::=    arrow «kind»₁ «kind»₂
```

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 α and β respectively and has output of type γ, should be. The type could be written `product α β → γ` (the `product` type is defined in the soon subsubsection "Product Type"), where `product α β` contains the two inputs. It could also be written `α → β → γ`, which associates to `α → (β → γ)`,

Sum Type The second notable primitive type is the *sum type*, where `sum α β` is the type of terms that are either of type α (exclusively) or of type β. This property is structured by the following declarations:

Listing 7.7: Primitives for the sum type

```
primitive type sum : kind → kind → kind.

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

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

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

Listing 7.8: Notation for infixed type sum

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

Notation for cases of a sum term. We also introduce the following syntax sugar for `split` — the following **`cases`** notation:

Listing 7.9: 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 7.1.2).

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 α, β, γ can be written `sum α (sum β γ)`. For example, the product of types α, β, γ 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 7.10: 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.

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 7.11: 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 7.10 is compatible with replacing the `case-`i with the constructors named by the named sum type definition since the constructors are basic terms.

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 α joined with a term of type β. This property is structured by the following declarations:

Listing 7.12: Primitives for `product`.

```
primitive type product : kind → kind → kind.

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

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

Listing 7.13: Notation for infixed type product

Infixed type product.

$\langle\!\langle kind \rangle\!\rangle_1 \times \langle\!\langle kind \rangle\!\rangle_2 \quad ::= \quad \texttt{product}\ \langle\!\langle kind \rangle\!\rangle_1\ \langle\!\langle kind \rangle\!\rangle_2$

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

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 α, β, γ can be written $\alpha \times (\beta \times \gamma)$. 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 7.14: Notation for constructing n-ary product terms

$(\langle\!\langle term \rangle\!\rangle_1, \ldots, \langle\!\langle term \rangle\!\rangle_n)$
$::=$
$\texttt{pair}\ \langle\!\langle term \rangle\!\rangle_1\ (\texttt{pair} \cdots (\texttt{pair}\ \langle\!\langle term \rangle\!\rangle_{n-1}\ \langle\!\langle term \rangle\!\rangle_n))$

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` $\circ$ `second` $\circ$ `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 7.15: Notations for destructing n-ary product terms

$\texttt{part-}i \quad ::= \quad (\texttt{second} \circ \overset{i-1}{\cdots} \circ \texttt{second} \circ \texttt{first})$

One other useful utility is the ability to apply a function to a particular component of the product, called *mapping* a function over that component. The following notation defines a collection of such mapping utilities.

Listing 7.16: Notations for mapping over n-ary product terms

$\texttt{map-}i \quad ::= \quad f\ (a_1, \ldots, a_n) \Rightarrow (a_1, \ldots, a_{i-1}, f\ a_i, a_{i+1}, \ldots, a_n)$

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 concise way of defining the type along with its appropriately named constructors and component maps.

Listing 7.17: Notation for defining named products types and utilities

```
type «type-name»* : «kind»*
  { «term-name»1:«type»1 ; ⋯ ; «term-name»n:«type-name»n }.

::=

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

// component destructors
term «term-name»1 : «type-name»* → «type»1 := part-1.
⋮
term «term-name»n : «type-name»* → «type»n := part-n.

// components maps ∀i
term map-«term-name»1 : («type»1 → «type»1) → «type-name»* → «type-name»*
  := map-1.
⋮
term map-«term-name»n : («type»n → «type»n) → «type-name»* → «type-name»*
  := map-n.
```

Lastly, the following notation provides an intuitive way to construct named product terms without having to remember the order of the components:

Listing 7.18: Notation for constructing named product terms

```
{ «term-name»1 := «term»1 ; ⋯ ; «term-name»n := «term»n }
  ::=
    («term»name1, …, «term»namen)
```

where $name_1, \ldots, name_n$ are the components named in the order intended for the underlying product type.

Other Common Primitives These are given in Appendix: Preludes.

Infixed-notation Associativity 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 $(\alpha \to \beta) \times (\gamma \oplus \delta)$

7.1.2.2 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 α and β, and forms the type `arrow α β`.

7.1.3 Typing

Table 7.2: Typing in Func

SIMPLE $\Gamma,\ a{:}\alpha \vdash a{:}\alpha$

SIMPLE $\Gamma,\ \alpha{:}A \vdash \alpha{:}A$

TERM-ABSTRACTION $$\frac{\Gamma,\ a{:}\alpha \vdash b{:}\beta}{\Gamma \vdash (a{:}\alpha) \Rightarrow b\ :\ \alpha \to \beta}$$

TERM-APPLICATION $$\frac{\Gamma \vdash f\ :\ \alpha \to \beta \qquad \Gamma \vdash a{:}\alpha}{\Gamma \vdash f\ a\ :\ \beta}$$

TYPE-ABSTRACTION $$\frac{\Gamma,\ \alpha{:}A \vdash \beta{:}B}{\Gamma \vdash (\alpha{:}A) \Rightarrow B\ :\ A \to B}$$

TYPE-APPLICATION $$\frac{\Gamma \vdash \varphi\ :\ \mathsf{A} \to \mathsf{B} \qquad \Gamma \vdash \alpha{:}\mathsf{A}}{\Gamma \vdash \varphi\ \alpha\ :\ B}$$

7.1.4 Reduction

Table 7.3: Reduction in Func

Simplify	$\dfrac{a \twoheadrightarrow a'}{a\ b \twoheadrightarrow a'\ b}$
Simplify	$\dfrac{b \twoheadrightarrow b' \qquad \text{value } v}{v\ b \twoheadrightarrow v\ b'}$
Apply	$\dfrac{\text{value } v}{((a : \alpha) \Rightarrow b)\ v \twoheadrightarrow [v/a]\ b}$
Split-Left	`split (left a) f g` $\twoheadrightarrow$ `f a`
Split-Right	`split (right b) f g` $\twoheadrightarrow$ `f b`
Project-First	`first (a, b)` $\twoheadrightarrow$ `a`
Project-Second	`second (a, b)` $\twoheadrightarrow$ `b`

7.2 Impe

7.2.1 Reduction

Table 7.4: Reduction in Impe

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

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

Table 7.5: Reduction in Impe: Sequencing

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

$$\textsc{Next}\quad \frac{\text{value } v}{\Delta \parallel v >> a \twoheadrightarrow \Delta \parallel a}$$

Table 7.6: Reduction in Impe: Mutability

$$\textsc{Initialize}\quad \frac{\text{value } v \qquad \mathsf{new}(\mathcal{S}) \rightsquigarrow (\mathcal{S}', id)}{\mathcal{S} \parallel \texttt{initialize}\ v \twoheadrightarrow \mathcal{S}'[\![id \mapsto v]\!] \parallel id}$$

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

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

Table 7.7: Reduction in Impe: Exception

$$\textsc{Throw}\quad \frac{\text{value } x}{\mathcal{E}[] \parallel \texttt{throw}\ e\ x \twoheadrightarrow \mathcal{E}[e,x] \parallel \eighthnote}$$

$$\textsc{Catch}\quad \mathcal{E}[e,x] \parallel \texttt{catch}\ \{\ e\ a \Rightarrow b\ \}\ \texttt{in}\ \eighthnote \twoheadrightarrow \mathcal{E}[] \parallel (a \Rightarrow b)\ x$$

$$\textsc{Valid}\quad \frac{\text{value } v}{\mathcal{E}[] \parallel \texttt{catch}\ \{\ e\ a \Rightarrow b\ \}\ \texttt{in}\ v \twoheadrightarrow \mathcal{E}[] \parallel v}$$

$$\textsc{Raise}\quad \mathcal{E}[e,x] \parallel \texttt{a}\ \eighthnote \twoheadrightarrow \mathcal{E}[e,x] \parallel \eighthnote$$

$$\textsc{Raise}\quad \mathcal{E}[e,x] \parallel \eighthnote\ \texttt{b} \twoheadrightarrow \mathcal{E}[e,x] \parallel \eighthnote$$

Table 7.8: Reduction in Impe: Nondeterminism

$$\textsc{Sample}\quad \mathcal{ND} \parallel \texttt{sample}\ l \twoheadrightarrow \mathcal{ND} \parallel \mathcal{ND}(\texttt{sample}\ l)$$

Table 7.9: Reduction in Impe: I/O

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

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

7.3 Mona

7.3.1 Reduction

Table 7.10: Reduction in Mona: I/O

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

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

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

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

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

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

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

7.4 Alge

7.4.1 Syntax

Table 7.11: Syntax for Alge

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

7.4.2 Typing

Table 7.12: Typing in Alge: Resource

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

Table 7.13: Typing in Alge: Channel

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

Table 7.14: Typing in Alge: Performance

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

Table 7.15: Typing in Alge: Handler

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

Table 7.16: Typing in Alge: Handling

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

7.4.3 Reduction

Table 7.17: Reduction in Alge

Simplify

$$\frac{\mathsf{monoperforms}(h) :: \mathcal{H}\ ;\ \mathcal{P} \parallel a \quad \twoheadrightarrow\ \mathsf{monoperforms}(h) :: \mathcal{H}\ ;\ \mathcal{P}' \parallel a'}{h :: \mathcal{H}\ ;\ \mathcal{P} \parallel a\ b \twoheadrightarrow h :: \mathcal{H}\ ;\ \mathcal{P}' \parallel a'\ b}$$

Simplify

$$\frac{\text{value } v \text{ relative to } \mathcal{H} \quad \mathsf{monoperforms}(h) :: \mathcal{H}\ ;\ \mathcal{P} \parallel b \quad \twoheadrightarrow\ \mathsf{monoperforms}(h) :: \mathcal{H}'\ ;\ \mathcal{P}' \parallel b'}{h :: \mathcal{H}\ ;\ \mathcal{P} \parallel v\ b \twoheadrightarrow h :: \mathcal{H}'\ ;\ \mathcal{P}' \parallel v\ b'}$$

Sequence

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

Handle

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

Perform

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

Handle-Final

$$\frac{\text{value } v \text{ relative to } \mathsf{unfinal}(h) \quad h := \texttt{handler}\ \{\ldots\ \texttt{final}\ b \Rightarrow c\ ;\ \ldots\}\ r}{\mathsf{unvalue}(h) :: \mathcal{H}\ ;\ \mathcal{P} \parallel v \twoheadrightarrow \mathcal{H}\ ;\ \mathcal{P} \parallel (b \Rightarrow c)\ v}$$

Handle-Value

$$\frac{\text{value } v \text{ relative to } h :: \mathcal{H} \quad h := \texttt{handler}\ \{\ldots\ \texttt{value}\ a \Rightarrow b\ ;\ \ldots\}\ r}{h :: \mathcal{H}\ ;\ \mathcal{P} \parallel v \twoheadrightarrow \mathsf{unvalue}(h) :: \mathcal{H}\ ;\ \mathcal{P} \parallel (a \Rightarrow b)\ v}$$

Handle-Performance

$$\frac{\text{value } w \text{ relative to } h :: \mathcal{H} \quad h := \texttt{handler}\ \{\ldots\ \#g\ x\ k \Rightarrow b\ ;\ \ldots\}\ r}{\mathsf{unvalue}(h) :: \mathcal{H}\ ;\ (r\#g\ v,\ @) :: \mathcal{P} \parallel w \quad \twoheadrightarrow\ \mathsf{unvalue}(h) :: \mathcal{H}\ ;\ \mathcal{P} \parallel (x\ k \Rightarrow b)\ v\ @(\ \Rightarrow w)}$$

Chapter 8

Appendix: Language Preludes

8.1 Func

8.1.1 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).
```

8.1.2 Data

Listing 8.1: unit

```
primitive type unit.

// only constructor
primitive term • : unit.
```

Listing 8.2: boolean

```
type boolean : Type { true | false }.
```

Listing 8.3: Notation for conditional

```
if «term»1 then «term»2 else «term»3
  ::=
    cases «term»1 { true  ⇒ «term»2
                  | false ⇒ «term»3
```

Listing 8.4: 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 8.5: 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

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

// addition; inherits notation
term add (i j : integer) : integer := ...

// subtraction; inherits notation
term sub (i j : integer) : integer := ...

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

Listing 8.6: rational

```
type rational : Type
  { numerator   : integer
  ; denomenator : 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).

term div (q r : rational) : rational := ...
```

Listing 8.7: 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
```

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

Listing 8.8: string

```
primitive type character.
primitive type string.

// 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.
```

Listing 8.9: Notations for characters and strings.

```
«term»_1 ++ «term»_2   ::=   string-concat [«term»_1, «term»_2]
```

8.1.3 Data Structures

Listing 8.10: optional

```
type optional (α : Type)
  { some : α → optional α
  | none : optioanl α }.
```

Listing 8.11: list

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

type list (α : Type) : Type
  { nil : list α
  | app : α → list α → list α }.
```

Listing 8.12: list notations

```
[]   ::=   nil

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

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

Listing 8.13: 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 }.

// tail of empty is empty
term tail (α : Type) (as : list α) : list α
  := cases as{ [] ⇒ [] | _ :: as' ⇒ as' }.

// undefined on out-of-bound indices
// gets the nth element of as
term index (α : Type) (as : list α) (n : natural) : α
  := cases n
       { zero    ⇒ cases as{ a :: _   ⇒ a }
       | succ n' ⇒ cases as{ _ :: as' ⇒ index as' n' } }.

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 replicate (α : Type) (n : natural) (a : α) : list α
  := cases n
       { zero    ⇒ []
       | succ n' ⇒ a :: repeat n' a }.
```

Listing 8.14: tree and forest

```
type tree (α : Type) : Type
  { value    : α
  ; branches : list (tree α) }.

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

Listing 8.15: 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.
```

Listing 8.16: 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.
```

Chapter 9

Mona

9.1 Type-Classes

9.1.1 Monad

Listing 9.1: Monad instances

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

instance Monad optional
  { lift   a  := some a
  ; map  f m  := cases m
                 { some a ⇒ some (f a)
                 | none   ⇒ none }
  ; bind m fm := cases m
                 { some a ⇒ fm a
                 | none   ⇒ none } }.

instance (α:kind) ⇒ Monad ((β:kind) ⇒ α → β)
  { lift    b := constant b
  ; map  f  m := f ∘ m
  ; bind m fm := a ⇒ fm (m a) a }.

instance (α:kind) ⇒ Monad ((β:kind) ⇒ α ⊕ β)
  { lift   b  := right b
```

```
  ; map  f m  := cases m
                 { left  a ⇒ left a
                 | right b ⇒ right (f b) }
  ; bind m fm := cases m
                 { left  a ⇒ left a
                 | right b ⇒ fm b } }.
```

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

9.1.2 Equalible

Listing 9.2: Equalible

```
class Equalible (α : kind) : kind
  { (==) : α → α → boolean }.
```

Listing 9.3: Equalible instances

```
instance Equalible unit := {⋯}.
instance Equalible boolean := {⋯}.
instance Equalible natural := {⋯}.
instance Equalible integer := {⋯}.
instance Equalible rational := {⋯}.

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

Bibliography

[1] Steven Awodey and Andrej Bauer. Propositions as [types]. *Journal of Logic and Computation*, 14:447–471, 08 2004.

[2] Andrej Bauer and Matija Pretnar. An effect system for algebraic effects and handlers. volume 10, 06 2013.

[3] Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *J. Log. Algebraic Methods Program.*, 84:108–123, 2015.

[4] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

[5] Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, 1940.

[6] Ishii H Kiselyov O. Freer monads, more extensible effects. volume 50, 03 2015.

[7] Robin Milner. Communicating and mobile systems: the pi-calculus. 1999.

[8] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55 – 92, 1991. Selections from 1989 IEEE Symposium on Logic in Computer Science.

[9] Simon Peyton Jones. *Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell*, pages 47–96. IOS Press, January 2001.

[10] Gordon Plotkin and John Power. Adequacy for algebraic effects. volume 2030, 01 2001.

[11] Gordon Plotkin and John Power. Notions of computation determine monads. volume 2303, 12 2001.

[12] Gordon Plotkin and Matija Pretnar. Handlers of algebraic effects. pages 80–94, 08 2009.

[13] Alan Turing. On computable numbers, with an application to the entscheidungsproblem. 1936.

[14] Philip Wadler. Propositions as types. *Communications of the ACM*, 58:75–84, 11 2015.