Chapter 1

Conventions

1.1 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.`"

1.2 Fonts

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

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

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

1.3 Names

Naming conventions:

[TODO] Finalize

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

Chapter 2

Introduction

[TODO] change code to codem in most cases, and have to wrap in $'s

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

[TODO] :

is this entire chapter, change concrete programs to be entirely monospace font

[TODO] change read and write to get and set

2.1 Introduction

[TODO] Introduce my thesis.

2.2 Language $\mathbb{A}$

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

2.2.1 Syntax for $\mathbb{A}$

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

Table 2.1: Syntax for 𝔸

metavariable	constructor	name
«*program*»	⟦ «*declaration*» ⟧	program
«*declaration*»	**primitive type** «*type-name*» : «*Type*».	primitive type
	type «*type-name*» : «*Type*» ≔ «*type*».	constructed type
	primitive term «*term-name*» : «*type*».	primitive term
	term «*term-name*» : «*type*» ≔ «*term*».	constructed term
«*Type*»	`Type`	atom
	`Type ->` «*Type*»	arrow
«*type*»	«*type-name*»	atom
	(«*type-param*» : «*Type*») ⇒ «*type*»	function
	«*type*» «*type*»	application
«*term*»	«*term-name*»	atom
	(«*term-param*» : «*type*») ⇒ «*term*»	function
	«*term*» «*term*»	application

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

- **primitive names**: Included a priori. Terms and types provided this way are provided along with their a priori type or Type respectively, and primitive terms along with a priori reduction rules (if any).
- **constructed names**: Require a definition written in the language.

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

- *non-terminals*: The meta-variables «*progam*», «*declaration*», «*Type*», «*type*», «*term*».
- *terminals*: Ranged over by the meta-variables «*term-name*», «*type-name*».
- *repeated*: Expressions of the form `[` «*meta-var*» `]` indicate that any number of «*meta-var*» can be repeated in its place.

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

- «*term-name*»: `1`, `12309`, `true`, `false`, `"hello world"`, `•`.
- «*type-name*»: `natural`, `integer`, `boolean`, `string`, `void`, `unit`.
- «*term-param*»: `x`, `y`, `this-is-a-parameter`.

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

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

2.2.2 Notations for 𝔸

Syntactical notations that translate definitional into their expansions before semantic processing are referred to as **syntactical sugar** in the programming community. In this way, they are *new* syntactical structures in 𝔸, but immediately reduce to the core set of syntactical structures as defined by the previously-outlined syntax of 𝔸. Since a few syntactical structures are used very commonly and are more intuitively read in a slightly different form, we shall adopt a few notations specified in the following few paragraphs.

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

⟦(«*term-param*»$_i$: «*type*»$_i$)⟧ «*term*»

::=

⟦(«*term-param*»$_i$: «*type*»$_i$) ⇒⟧ «*term*»

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.

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

::=

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

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

::=

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

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

::=

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

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

Listing 2.1: Notation for multiple parameters.

```
⟦(«term-param»_i : «type») ⇒⟧

::=

(⟦«term-param»_i⟧ : «type») ⇒
```

Listing 2.2: Notation for consecutive shared-typed parameters.

⟦(«*type-param*»$_i$: «*Type*») ⇒⟧

::=

(⟦«*type-param*»$_i$⟧ : «*Type*») ⇒

Local bindings. These are a core feature in all programming languages, and is expressed in 𝔸 with this notation:

Listing 2.3: Notation for local binding.

```
let «term-param» : «type» ≔ «term»₁ in «term»₂

::=

(«term-name» : «term» ⇒ «term»₂) «term»₁
```

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`.

2.2.3 Primitives for 𝔸

[TODO] rewrite these with syntax sugar

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 its `construct term` and `constructed type` declarations. There are two particularly useful primitive types, along with some accompanying primitive terms, to have defined as part of the core of 𝔸. The first of these is the `sum` type, where `sum α β` is the type of either `α` or `β`. To reflect this, a term `x : sum α β` is constructed with either a term of type `α` or a term of type `β`. These properties are specified by the following declarations:

Listing 2.4: Primitives for `sum`.

```
primitive type sum : Type → Type → Type.

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

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

Notation 2.2.1. Case of sum.

[TODO] review this notation

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

Listing 2.5: Notation for `case`.

```
case «term» to «type»
  { left («term-param»₁:«type»₁) ⇒ «term»₁
  ; right («term-param»₂:«type»₂) ⇒ «term»₂ }

::=

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

[TODO] Explain n-ary sums.

Notation 2.2.2. [TODO] Notation for constructing and destructing n-ary sums.

The next notable primitive type to define here is the `product` type, where `product α β` is the type of both α and β. To reflect this, a term `x : product α β` is constructed with both a term of type α and a term of type β. These properties are reflected by the following declarations:

Listing 2.6: Primitives for `product`.

```
primitive type product : Type → Type → Type.
```

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

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

[TODO] Explain n-ary products

Notation 2.2.3. [TODO] Notation for constructing and destructing n-ary products.

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

- integer
- natural
- boolean
- unit

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

Notation 2.2.4. Infix arrow, sum, and product types. The arrow, sum, and product types have a usual infixed notation to enhance the readability as they are immensely common and intentionally intuitive. These notations are as follows:

Listing 2.7: Notations for infixed arrow, sum, and product types

```
«type»₁ → «type»₂    ::=   arrow «type»₁ «type»₂
«type»₁ + «type»₂    ::=   sum «type»₁ «type»₂
«type»₁ × «type»₂    ::=   product «type» «type»
```

Each of these infixed notations are right-associative. So, the type $\alpha \to \beta \to \gamma$ implicitly expands to $\alpha \to (\beta \to \gamma)$, matching a Curry-oriented[1] intuition.

[1] TODO: Citation needed?

Additionally, having multiple infixed notations for «*type*»s introduce inter-notational ambiguity. For example, the type $\alpha \to \beta \times \gamma + \delta$ could be associated in many different ways. We adopt the following precedence order of increasing tightness:

$$\to, +, \times.$$

According to this order, the type $\alpha \to \beta \times \gamma + \delta$ expands to the proper association $\alpha \to ((\beta \times \gamma) + \delta)$.

2.2.4 Type application.

The notation for «*type-name*» «*type*» ⋯ «*type*» 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`. Note that this is a different kind of function than the **function** syntactical construct, but the intuitive similarity will be addressed later in chapter **??**.

2.2.5 Typing Rules for 𝔸

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

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

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

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

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

The following typing rules are given for 𝔸:

[**TODO**] describe Typing rules for Types (of types)

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

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

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

2.2.6 Reduction Rules for 𝔸

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

Table 2.2: Reduction in 𝔸

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

$$\textsc{Simplify-Application}\ \frac{a \twoheadrightarrow a' \qquad \text{value } v}{a\ v \twoheadrightarrow a'\ v}$$

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

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

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

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

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

[TODO] note that APPLY is usually referred to as β-reduction.

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, consider the following terms:

[TODO] enlightening example of a series of β-reductions for some simple computation.

2.2.7 Properties of $\mathbb{A}$

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

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

Theorem 2.2.1. (Type-Preserving Substitution in $\mathbb{A}$). If $\Gamma, a{:}\alpha \vdash b{:}\beta$, $\Gamma \vdash v{:}\alpha$, and value v, then $\Gamma \vdash ([v/a]b){:}\beta$.

Theorem 2.2.2. (Reduction Progress in $\mathbb{A}$). If $\{\} \vdash a{:}\alpha$, then either value a or there exists a term a' such that $a \twoheadrightarrow a'$.

Theorem 2.2.3. (Type Preservation in $\mathbb{A}$). If $\Gamma \vdash a{:}\alpha$ and $a \twoheadrightarrow a'$, then $\Gamma \vdash a'{:}\alpha$.

Theorem 2.2.4. (Type Soundness in $\mathbb{A}$). If $\Gamma \vdash a{:}\alpha$ and $a \twoheadrightarrow a'$, then either value a' or there exists a term a'' such that $\Gamma \vdash a''{:}\alpha$ and $a' \twoheadrightarrow a''$.

Theorem 2.2.5. (Strong Normalization in $\mathbb{A}$). For any term a, either value a or there is a sequence of reductions that ends in a term a' such that value a.

Chapter 3

A Simple Approach to Effects

[TODO] add captions to all listings

3.1 Computation with Effects

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

The definition of 𝔸 in section ?? embodies a good flavor for many similar declarative programming languages. In terms of the such language's reductions from term to term, all the information relevant for deciding such reductions is explicit within the term itself and the explicit context built up during reduction. In other words, there is no **implicit activity** that influences what a term's reduction will look like. The path of reductions is exactly the computation a term corresponds to, so this yields that the computation has the same property of not having access to or being affected by any implicit activity.

However nice a formalization of computation this is, it is immediately unrealistic. Actual computers, for which programming languages are abstractions of their activities, host multitudes of implicit processes while running a program. Even if a programming language modelled all such processes and incorporated them into the language so that each activity was made perfectly explicit as a term (a task that is certainly infeasable and likely impossible), the result would be an unuseful and inefficient language. The point of having layers of abstraction, in the form of high-and-higher level programming language, is to avoid this situtation in the first place. So there appears to be a dilemma:

The Dilemma of Implicit Activities

(1) Require fully explicit terms and reductions. This grants reasoning about programs is fully formalized and abstracted from the annoyances of hardware and lower-level-implementations, but restricts such programs from being applicable in almost all useful circumstances.

(2) Allow implicit activities 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 inescapably tainted by implicit activities.

A resolution to this apparent dilemma is to either choose one horn or to reject the dilemma. But first, let us consider what kinds of implicit activities are being considered here — in particular, what are computational **effects**.

3.1.1 Effects

The definition of an **effect** in the context of programming language is frequently debated, and there is no majority standard answer[1]. For the purposes of this thesis, the following definition is adopted.

Definition 3.1.1. A computational **effect** is a capability in a program that depends on factors outside of that capability's normal scope.

The definition of **normal scope** will be left to intuitive interpretation, with the intent that what is the normal scope of a capability depends on many theoretical, design, and implementation factors.

Now for example, suppose we have a program P that computes the sum of two integers given as input. If P is designed and implemented exactly to this specification, then P has no effects; none of P's cabapilities depend on factors outside of their usual scope. The only scope that P's capabilities depend on is that of the two inputs. No other factors influence what the correctly computed sum of the two integers is. Such a program with no effects is called a **pure** program.

Definition 3.1.2. A program is **pure** if it has no effects.

But if P is run, as abstractly as it is defined, not much use comes from it. P does not display its results, write the results to some P-specified memory, or give you an error if there is an overflow. These capabilities depend on factors outside of P's normal scope, per its specification, and so are examples of effects.

So as another example, let us consider a modified version of P that is effectual. Suppose that program P' takes as input two integers, computes the sum of the integers, throws an error if there is an overflow, writes the result into RAM, and prints the result to the console from which P' was run. These capabilities are examples of effects, since their behavior depends on factors outside of the normal scope of P' (the same scope as P). Such a program with effects is called an **impure** program.

Definition 3.1.3. A program is **impure** if it has effects.

[1] [TODO] source

There are a variety of common effects that are available in almost every programming language. These include:

- **input/output (IO),** e.g. printing to the console, accepting input from use, interfacing with other peripherals.
- **mutability**, e.g. mutable variables, in-place arrays.
- **exception**, e.g. division by `0`, out-of-bounds index of array, head of empty list.
- **nondeterminism**, e.g.

  [**TODO**] what would be good examples for this…

  [**TODO**] go into detail here, or till after I've talked about comparing declarative/imperative approaches to effects?

According to definition 3.1.1, there is an easy method for transforming an impure program into a pure program: add the factors depended upon by the program's effects to the program's scope. Using this intuition, we can reformulate the Dilemma of Implicit Activities with the new formal notion of computational effects:

The Dilemma of Effectual Purity

(1) Require only pure programs. This grants reasoning about programs to depend only on the normal scope of the program.

(2) Allow impure as well as pure programs. This grants many useful programs, where the behavior of the programs depends on factors not entirely encapsulated by the program's normal scope.

The goal of resolving this dilemma is to make a choice about how programming languages should be designed. Are computational effects a feature to be avoided as much as possible? Or are they actually so necessary that it would be a mistake to restrict them? Unsurprisingly, no widely-adopted languages have chosen horn (1). But even in more generality, almost all languages have chosen horn (2), but with many varied approaches to incorporating effects. Overall, languages have clustered into two groups.

- **Declarative** programming language treat computation as the evaluation of mathematical functions. Such languages put varying degrees of emphasis on restricting effects, but in general much more emphasis than imperative languages. E.g. Scheme, Lisp, Standard ML, Clojure, Scala, Haskell, Agda, Gallina.
- **Imperative** programming languages treat computation as a sequence of commands. E.g. the C family, Bash, Java, Python. Such languages put very little (if any) emphasis on restricting effects.

The perspective on effects is not the only way in which these groups differ, but nevertheless it is an important one. Among declarative programming languages, we shall consider two in particular: Standard ML and Haskell. Standard ML takes a very relaxed approach to effects, and Haskell takes a more restrictive approach.

> [**TODO**] summarize a small history lesson of how there were two camps in the approach to effects: C code (systems-level programming) and PL people (type theory e.g. ML). For most of history, C code has won out, but more recently the advantages of structures in the lambda-calculus and type theory have started making their way into industry languages (e.g. Java has generics and lambdas, Haskell is getting mode industry use).

3.2 Definition of 𝕀𝔹

The Standard ML (Standard Meta Language) language is most commonly used in academic settings for teaching about programming languages. However here, I shall avoid introducing an entire new syntax and language semantics. Instead, the Standard ML design for effects will be demonstrated using 𝕀𝔹, where 𝕀𝔹 extends 𝔸𝕃 with the a few new effect-related features.

3.2.1 Primitives for 𝔹

[TODO] write English to encompass this

Sequencing

Since effects are context-sensitive in a way visible from within 𝔸, we'd like 𝔹 to allow express, explicitly, the order in which some effects are to be performed. Such an expression is an application of *sequencing*. A primitive term allows sequences to be written, and 𝔹's reduction rules mirror them by reducing terms in the correct order.

[TODO] describe

Listing 3.1: Definition for sequencing

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

Listing 3.2: Notation for infixed sequence

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

Mutability

Most imperative programming languages rely extensively on the ability to assign values to variables that are allowed to be recalled and mutated later on in the program. Declarative programming languages often lack this feature and rely instead on immutable data. A simple way of introducing mutability to a declarative language like 𝔸 is to posit a data type that wraps a value, and an effectual interface for mutating and getting the wrapped value.

This wrapper data type is encoded as the `mutable` type, which is parametrized by the type of the data it stores. 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 in a given mutable, and the term `set` sets to a new given value the value stores in a given mutable.

A perhaps familiar conceptualization is to think of a term `ma : mutable` α is similar to an α-*pointer* in C (in fact, some of the following notation was chosen to take advantage of this intuition). In this way, `ma` is a reference to some memory, `get` reads the referenced memory, and `set` updates the referenced memory. However, `mutable` is more general than pointers in C because it is abstracted away from any specific implementation of such memory reference — it could be that `mutable` uses pointers or it could be that it uses only immutable memory.

Listing 3.3: Definitions for mutablility

```
primitive type mutable : Type → Type.

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

Listing 3.4: Notations for mutability.

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

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

«term»_1 <- («term»_2 : «type»)    ::=    set «type» «term»_1 «term»_2
```

Exceptions

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

One common way of anticipating partiality is to introduce *exceptions*, which are specially-defined ways for a function to result when it is applied to certain otherwise-undefined inputs. The immediate issue with extending 𝔸 with naive exceptions is that it requires exceptions to be of the same type as the expected result. Schematically, our division function example looks like this:

```
term division (i j : integer) : integer
  ≔ if j == 0
      then throw an exception
      else divide as usual
```

Here, the result of whatever is implemented in place of *"throw exception"* must yield an integer. However, this amounts to delegating a somewhat-arbitrary result in place of undefined results[2].

[2]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.

A simple declarative-friendly way to allow implicitly-exceptional results is to introduce a term that uses an exception instance to produce a stand-in term of any type. Instances of `excepting 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 of type `(excepting 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 : β`, then `catching` results in the continuation, supplied with the specific thrown exception and its argument, instead of continuing to evaluate `b`.

So in 𝕀𝔹 with this exception framework, we can schematically write the division function like this:

```
primitive type division-by-0 : excepting integer.

term division (i j) : integer
  ≔ if j == 0
      then throw division-by-0 j
      else divide as usual
```

The term `throw division-by-0 j` will be typed as an `integer`, but during evaluation it should not exactly yield an integer. Instead, some outer `catching` application must specify how to deal with `division-by-0` throws.

Listing 3.5: Definitions for exception

```
primitive type excepting : Type → Type.

primitive term throw (α β : Type) : excepting α → α → β.
primitive term catching (α β : Type)
  : (excepting α → α → β) → β → β.
```

Listing 3.6: Notation for exception

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

::=

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

Input/Output (IO)

Finally, IO 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 IO within our language must be very general. As an easy setup, we shall use an IO interface that just deals with `strings`. However, it is easy to imagine other specific IO functions that would work in a similar vein (e.g. `input-integer`, `input-time`, `output-image`, etc.).

The term `input` obtains a string through the IO interface,

Listing 3.7: Definitions for IO

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

3.2.2 Reduction Rules for 𝔹

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

[**TODO**] explain how S and E are handled when not included in inference rule (stay same)

Reduction Rules for Sequences

leftoff

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

[**TODO**] Define

Reduction Rules for Exceptions

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

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

Table 3.1: Reduction in 𝔹: Exceptions

$$\textsc{Throw}\quad \frac{e : \texttt{excepting}\ \alpha \qquad v : \alpha \qquad \text{value } v}{\mathcal{E} \parallel \texttt{throw}\ e\ v \twoheadrightarrow \texttt{throw}\ e\ v :: \mathcal{E} \parallel \texttt{throw}\ e\ v}$$

$$\textsc{Raise}\quad \frac{e : \texttt{excepting}\ \alpha \qquad \text{value } v}{\texttt{throw}\ e\ v :: \mathcal{E} \parallel \texttt{e} \twoheadrightarrow \texttt{throw}\ e\ v :: \mathcal{E} \parallel \texttt{throw}\ e\ v}$$

$$\textsc{Catch}\quad \frac{e : \texttt{excepting}\ \alpha \qquad \text{value } v}{\texttt{throw}\ e\ v :: \mathcal{E} \parallel \texttt{catch}\ \{\ e\ a \Rightarrow b\ \}\ \texttt{in}\ v}$$

TODO: more

Reduction Rules for IO

Table 3.2: Reduction in 𝔹: IO

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

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

These rules interact with the IO context, $\mathcal{O}$, by using it as an interface to an external IO-environment that handles the IO effects. This organization makes semantically explicit the division between 𝔹's model and an external world of effectual computations. For example, which $\mathcal{O}$ may be thought of as stateful, this is not expressed in the reduction rules as showing any stateful update of $\mathcal{O}$ to $\mathcal{O}'$ when its capabilities are used. An external implementation of $\mathcal{O}$ that could be compatible with 𝔹 must satisfy the following specifications:

Specification of $\mathcal{O}$

[**TODO**] should $\mathcal{O}$(`output `) be specified as anything?

- $\mathcal{O}$(`input •`) returns a `string`.
- $\mathcal{O}$|(`output s`) returns nothing, and resolves to $\mathcal{O}$.

[**TODO**] come up with running IO example

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

Listing 3.8: Hello World

```
output "hello world"
```

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

Implementation 1 of $\mathcal{O}$:

- $\mathcal{O}$(`input •`):

  1. Prompt the console for user text input.

2. Interpret the user text input as a string, then return the string.

- `𝒪(output s)`:

  1. Write `s` to the console.
  2. Resolve to `𝒪`.

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

Implementation 2 of `𝒪`:

- `𝒪(input •)`:

  1. Set the toaster periphery's mode to **currently toasting**.

- `𝒪(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 to `𝒪`.

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

3.3 Motivations

This chapter has introduced the concept of effects in programming languages, and presented 𝕀𝔹 as a simple approach to extending a simple lambda calculus, 𝔸\, with a sample of effects (mutable data, exceptions, IO). But such a simple approach leaves a lot to be desired.

Each effect is implemented by pushing effectual computation to the reduction context:

- Mutable data is managed by a mutable mapping between identifiers and values.
- Exceptions are thrown to and caught from an exception stack.
- IO is performed through an interface to an external IO implementation.

  [TODO] describe problem with this kind of approach

- mutable data is global and stored outside language objects

- exceptions bypass type checking; exceptable programs aren't reflected in types
- IO is a black-box which is not reflected in types, so cannot be modularly reasoned about in programs (similar to exceptions)

Chapter 4

Monadic Effects

[TODO] add captions to all listings

[TODO] change read and write to get and set

4.1 Introduction to Monads

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

The concept of *monad* originates in the branch of mathematics by the name of Category Theory. As highly abstract as monads are, they turn out to be a very convenient structure for formalizing implicit contexts within functional programming languages. Though a background in the Categorical approach to monads and monad-related structures probably cannot hurt one's understanding of computational monads, this section will follow a more programmer-friendly path.

The Stateful Monad

One particularly general effect is the **stateful** effect, where an implicit, mutable state is accessible within a stateful computation. This effect was implemented by 𝔹 by the introduction of a globally-accessible, mutable table of variables. This impl required, however, several new syntactical structures and reduction rules. Is there a way to implement something similar in just 𝔸\?

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

```
type stateful (σ a : Type) : Type ≔ σ → σ × α.
```

Relating to the description of the stateful effect, `stateful σ α` is the type of functions from an initial σ-state to a pair of the affected σ-state and the α-result. So if given a term m : `stateful σ α`, one can purely compute the affected state and result by providing m with an initial state.

For `stateful` to truly be an effect, it needs to implicitly facilitate the stateful effect to some sort of internal context. So let us see how we can construct terms that work with `stateful`. In 𝔹's impl of this effect, it posited two primitive terms: `read` and `write`. Using `stateful` we can define these terms in 𝔸 as:

```
term read {σ : Type}
  : stateful σ σ
  ≔ (s : σ) ⇒ (s, s).

term write {σ : Type} (s' : σ)
  : stateful σ σ
  ≔ (s : σ) ⇒ (s', •).
```

Observe that `read` is a σ-computation that does not modify the state and lifts the state, `write` is a σ-computation that replaces the state with a given $s' : \sigma$ and lifts $\bullet$. In this way, using `read` and `write` in 𝔸 fills exactly the same role as a simple 𝔹 stateful effect of one mutable variable.

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

Since the stateful 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 sequencing of just two effects, since any number of effects can be sequenced one step at a time. So, given two σ-computations m : `stateful σ α` and m' : `stateful σ β` the sequenced σ-computation should first compute the m-affected state and then pass it to m'.

```
term stateful-sequence
      (σ α : Type)
      (m : stateful σ α) (m' : stateful σ β) :
  : stateful σ β
  ≔ (s : σ) ⇒
      let (s', a) : σ × α ≔ m s in
      m' s'.
```

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

```
term stateful-bind
      (m : stateful σ α)
      (fm : α → stateful σ β)
  : stateful σ β
  ≔ (s : σ) ⇒
      let (s', a) : σ × α ≔ m s in
      fm a s'.
```

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

```
term stateful-sequence
      (σ α β : Type)
      (m : stateful σ α) (m' : stateful σ β)
  : stateful σ β
  ≔ stateful-bind m ((a : α) ⇒ m')
```

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

So far, we have defined all of the *special* operations that 𝔹 provides for stateful computations. The key difference between 𝔹 and our new 𝔸 impl of the stateful effect is that 𝔹 treats stateful computations just like any other kind of pure computation, whereas 𝔸 "wraps" σ-stateful computations of α using the `stateful` $\sigma\ \alpha$type rather than just pure α. What is still missing in our 𝔸 impl is the ability to use non-stateful computations within or on stateful computations. In other words, we should be able to *map* non-stateful computations to stateful computations that internally don't actually end up having stateful effects. There are two such computations to consider:

[TODO] describe these a little more intuitively with more detail

- **lift**: The non-stateful computation, with parameter $a : \alpha$, that results in the same $a : \alpha$. In other words, the computation that does nothing and results in a.

- **map**: The computation, with parameter non-stateful computation $f : \alpha \to \beta$ and stateful computation m : `state` σ α, that results in f applied to the result of m. In other words, the function the maps f : $\alpha \to \beta$ to a function `stateful` σ β → `stateful` σ β.

These computations are implemented in 𝔸 as follows:

```
term stateful-lift (a : α) : stateful σ α ≔ (s : σ) ⇒ (s, a)
```

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

4.1.1 Formalizion of Monad

Definition of Functor

[TODO] explain how this is useful for monad. Also, monad instances are also functor instances.

Listing 4.1: Definition of the Functor type-class

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

Definition of Monad

Now we have implemented a concrete approach to the stateful effect using pure terms in 𝔸. However, only the `read` and `write` terms were meant for exclusive use with `stateful`. The other necessary terms, as they correspond to use the impure implementation in 𝔹, are intended to interoperate with all effects and interchangeably. To summarize, the capabilities the terms implement are

- **map**
- **lift**

- **binding-sequence**

With these in mind, the definition of monad is as follows:

Definition 4.1.1. A type $M : Type \to Type$ is a **monad** if there exist terms of the following types:

- `map : (α → β) → M α → M β`
- `lift : α → M α`
- `bind : M α → (α → M β) → M β`

However, within 𝔸 we currently have no type-oriented way to assert that a type M is associated with the expected constructions for qualifying as a monad. In order to formally and generally reference monads in 𝔸, we first need to introduce the concept type-classes.

4.2 Type-classes

4.2.1 Concept of Classes

The concept of a *class* is used in many different forms among many different programming languages. In object-oriented programming languages, classes define "blueprints" for creating objects that are instances of the class. Such a class defines an *interface* that each instance object of the class must implement. For example, the following Java code defines the abstract `Animal` class as the class of objects that have a `name` and implement the `eat` and `sleep` functions:

```
abstract class Animal {

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

  abstract public void eat();
  abstract public void sleep();

}
```

```
class Animal {
```

```
  public void eat() {
    System.out.println(this.name + " eats.");
  }

  public void sleep() {
    System.out.println(this.name + " sleeps.");
  }

}
```

The motivation for the classes is to allow a parameter to range over a collection of different types of structures given that all of these structures meet some requirements. Then the parameter can be assumed to meet those requirements, regardless of the specific argument structure ends up being provided.

As is commonly known, both cats and dogs are examples of animals, so we can create respective objects that are instances of the `Animal` class:

```
class Cat extends Animal {

  public Cat() {
    super("Cat");
  }

  @Override
  public void eat() {
    System.out.println(this.name + " eats kibble.");
  }

  @Override
  public void sleep() {
    System.out.println(this.name + " naps.");
  }

  public void hunt() {
    System.out.println(this.name + " hunts for some mice and birds.");
  }

}

class Dog extends Animal {

  public Dog() {
```

```
    super("Dog");
  }

  @Override
  public void eat() {
    System.out.println(this.name + " eats steak.");
  }

  @Override
  public void sleep() {
    System.out.println(this.name + " sleeps wrestlessly.");
  }

  public void walk() {
    System.out.println(this.name + " goes on a walk with its owner.");
  }

}
```

Finally, and most importantly, we can write functions that take `Animal` parameters and only assume of these parameters what is specified by the `Animal` class — even though `Cat` and `Dog` are different structures.

```
void simple_simulate_animal(Animal a, int steps) {
  for (int i = 0; i < steps; i++) {
    a.eat();
    a.sleep();
  }
}
```

4.2.2 Definition of type-classes

Now back to the realm of strictly-typed functional programming languages. Our motivation at the moment is to devise a structure, describable in 𝔸, that allows for the formal specification of the monad and other type-class.

It turns out that we can model the Monad type-class as a parametrized type, with the type `Monad (M:Type) : Type` of terms that implement the monad requirements for M. In other words, a term of type `Monad M` "contains" in some way (i.e. *implements*) terms with the types of `map`, `lift`, and `bind` as defined in section 4.1.1. The simplest way to represent such an *implementation* of a type-class instance in this way is to represent the implementation as a type-product of the types of the required terms. Concretely, for the `Monad` type-class, we specify the `Monad` type as

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

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

```
term map (M : Type → Type) (α β : Type) (impl : Monad M)
  : (α β : Type) ⇒ (α → β) → M α → M β
  := first impl

term lift (M : Type → Type) (α : Type) (impl : Monad M)
  : (α : Type) ⇒ α → M α;
  := second impl

term bind (M : Type → Type) (α, β : Type) (impl : Monad M)
  : (α, β : Type) ⇒ M α → (α → M β) → M β
  := third impl
```

[TODO] define notation for these, such as » and »= and let <- in

Listing 4.2: Notations for binding.

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

```
let «term-param» <- «term»₁ in «term»₂   ::=   bind «term»₁ («term-param»
    => «term»₂)
```

Listing 4.3: Notation for sequencing.

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

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

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

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

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

Notations for type-classes

So far, the code that we have ended up writing to define a type-class and implement an instance of that type-class has contained a lot of boilerplate and is hard to follow on its own. For example, the types of `map`, `lift`, and `bind` had to be specified twice: once in the specification of `Monad`, and once again in their `Monad`-general constructions. To make the definition and instantiation of type-classes easier to read and write, we introduce the following notations.

Notation 4.2.1. Type-class definition. Defining a type-class requires specifying the terms that must be implemented for each instance of the type-class. The following notation allows concise specification, as well as additionally generating the terms, generalized to all instances of the type-class, that are specified.

Listing 4.4: Notation for type-class definition

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

::=

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

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

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

Notation 4.2.2. Type-class instantiation. Instantiating the a type A as an instance of a type-class C requires implementing the terms specified by C where A is supplied as the argument to C's first type parameter. The following notation conveniently names this implementation and makes explicit C's intended names for each component.

Listing 4.5: Notation for type-class instantiation

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

::=

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

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

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

Listing 4.6: Instance of the stateful monad

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

Notation 4.2.3. Type-class passive status.

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

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

[TODO] Give definitions of other monads that implement other effects, such as exception and IO (and maybe others?)

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

Chapter 5

Algebraic Effect Handlers

[TODO] the way that performance wraps values is not good. need some sort of bowtie-like semantics to make it work

5.1 Introduction to Algebraic Effect Handlers

In chapter 3, we considered 𝔹, an extension of 𝔸, that implemented effects by introducing specific language features for each kind of effect. While this allows simple reasoning about the behavior of those effects in 𝔹, it establishes no common reasoning about effects in general. If 𝔹 were extended to implement another effect using a new language feature, none of what is defined in 𝔹 explains how it should be implemented how it might behave.

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

- Effects may be defined as "black boxes" with implicit behavior defined outside the language. For example, the *IO* effect must appeal to an *IP* interface at some point, just like in 𝔹.
- Effects may be defined completely within the language. For example, the *state* and *exception* effects can be compeltely modeled within the language (e.g. chatper 4).

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

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

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

- **Performances:** The code that corresponds to the *performing* of an effect. Performances are sensative to the whole program context.

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

Additionally, this setup requires an interface to the effects that are to be performed and handled.

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

5.2 Language ℂ

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

5.2.1 Syntax for ℂ

Table 5.1: Syntax for ℂ

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

5.2.2 Primitives for ℂ

Resource

[TODO] description

Listing 5.1: Primitives for resources.

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

Action

[TODO] description

Listing 5.2: Primitives for actions.

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

Listing 5.3: Notation for action.

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

Performance

[TODO] description

Listing 5.4: Primitives for performance.

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

Listing 5.5: Notation for performance.

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

Handling

[TODO] description

[TODO] note that ⊳ is for “mapping”

[TODO] note the significance of the handler being the first argument, which is evaluated last during reduction

Listing 5.6: Primitives for handling.

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

primitive term make-handler (α β γ : Type) (ρ : Resource)
  : list ((χ ↗ υ) × ((χ × (α → β)) → β)) ×
    (α → β) ×
    (β → γ) ×
    ρ →
    handling α γ.

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

Listing 5.7: Notation for handling.

«*type*»$_1$ ↘ «*type*»$_2$ ::= handling «*type*»$_1$ «*type*»$_2$

Listing 5.8: Notation for making a handler.

```
handler
  { ⟦ #«action-name»_i «term-param»_i1 «term-param»_i2 ⇒ «term»_i ;⟧
  ; value «term-param»_2 ⇒ «term»_2
  ; final «term-param»_3 ⇒ «term»_3 }

::=

make-handler
  [⟦ «action-name»_i ↦ («term-param»_i1 «term-param»_i2 ⇒ «term»_i) ,⟧]
  («term-param»_2 ⇒ «term»_2)
  («term-param»_3 ⇒ «term»_3)
```

Listing 5.9: Notations for doing handling with.

with «*term*»$_1$ do «*term*»$_2$ ::= handle «*term*»$_1$ «*term*»$_2$

do «*term*»$_1$ with «*term*»$_2$ ::= handle «*term*»$_2$ «*term*»$_1$

Sequencing

[TODO] description

[TODO] note: don't need binding since effects aren't typed

Listing 5.10: Primitives sequencing.

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

[TODO] Note: uses same notations as introduced in monadic-effects chapter. Should I introduce these notations in the simple-effect chapter instead? Or would it not work there...

5.2.3 Typing Rules in $\mathbb{C}$

Table 5.2: Typing in $\mathbb{C}$

$$\textsc{Resource}\quad \frac{\rho \coloneqq \texttt{resource\{}\ [\![\ e_i\texttt{:(}\alpha_i \nearrow \beta_i\texttt{)}\ \texttt{;}]\!]\ \texttt{\}}}{\Gamma \vdash \texttt{(resource\{}\ [\![\ e_i\texttt{:(}\alpha_i \nearrow \beta_i\texttt{)}\ \texttt{;}]\!]\ \texttt{\}):Resource}}$$

5.2.4 Reduction Rules for $\mathbb{C}$

[**TODO**] Since Handle-Effect only works on statements that aren't the *last* effect, there's a notation that appends a trivial ending to anything of that form.

[**TODO**] need to rephrase these in terms of pushing the handlers onto a stack since can have nested handlers

[**TODO**] Explain how contextual value judgement works. Since the presence or absense of a certain handler in the handler context can change whether or not a term is reducible.

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

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

"value v with $\mathcal{H}$" is true if and only if value v and there is no handler h among the $\mathcal{H}$ such that h has a `value` clause.

[**TODO**] Should the priority of Raise be before of after Perform - Handle-Performance?

Table 5.3: Reduction in $\mathbb{C}$

Simplify

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

Simplify

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

Raise

$$\frac{\mathcal{H} \;;\; \mathcal{P} \parallel a \;\twoheadrightarrow\; \mathcal{H}' \;;\; \mathcal{P}' \parallel a' \qquad \text{value } a \text{ with } [h]}{h :: \mathcal{H} \;;\; \mathcal{P} \parallel a \;\twoheadrightarrow\; h :: \mathcal{H}' \;;\; \mathcal{P}' \parallel a'}$$

Do

$$\mathcal{H} \;;\; \mathcal{P} \parallel \texttt{do}\ a\ \texttt{with}\ h \;\twoheadrightarrow\; h :: \mathcal{H} \;;\; \mathsf{finish}(h) :: \mathcal{P} \parallel a$$

Perform

$$\frac{p := r\#e\ v \qquad \text{value } v}{\mathcal{H} \;;\; \mathcal{P} \parallel p \;\twoheadrightarrow\; \mathcal{H} \;;\; (r, e, v, a) :: \mathcal{P} \parallel a}$$

Handle-Final

$$\frac{h := \texttt{handler}\ \{\ldots\ \texttt{final}\ b \Rightarrow c \;;\; \ldots\}\ \mathsf{r} \qquad \text{value } v \text{ with } h :: \mathcal{H}}{h :: \mathcal{H} \;;\; \mathsf{finish}(h) :: \mathcal{P} \parallel v \;\twoheadrightarrow\; \mathcal{H} \;;\; \mathcal{P} \parallel (b \Rightarrow c)\ v}$$

Handle-Value

$$\frac{\begin{array}{l} h := \texttt{handler}\ \{\ldots\ \texttt{value}\ a \Rightarrow b \;;\; \ldots\}\ \mathsf{r} \\ h' := h \text{ without } \texttt{value} \text{ clause} \\ \text{value } v \text{ with } h :: \mathcal{H} \end{array}}{h :: \mathcal{H} \;;\; \mathcal{P} \parallel v \;\twoheadrightarrow\; h' :: \mathcal{H} \;;\; \mathcal{P} \parallel (a \Rightarrow b)\ v}$$

Handle-Performance

$$\frac{h := \texttt{handler}\ \{\ldots\ \#e\ x\ k \Rightarrow b \;;\; \ldots\}\ \mathsf{r} \qquad \text{value } v \text{ with } h :: \mathcal{H}}{\begin{array}{l} h :: \mathcal{H} \;;\; (r, e, w, a) :: \mathcal{P} \parallel v \;\twoheadrightarrow \\ h :: \mathcal{H} \;;\; \mathcal{P} \parallel (x\ k \Rightarrow b)\ w\ (a \Rightarrow v) \end{array}}$$

5.3 Examples

5.3.1 Example: Nondeterminism

Resource

[TODO] describe

Listing 5.11: Resource for nondeterministic coin-flipping.

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

// create a new resource instance of the coin-flipping effect
term coin : Coin := new Coin.
```

Experiment

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

Listing 5.12: Experiment with nondeterministic coin-flipping.

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

Handling all possible flips

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

Listing 5.13: Handler for either heads or tails.

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

Listing 5.14: Do experiment with either heads or tails.

```
term h ≔ either-heads-or-tails coin.

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

Handling singular flips

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

Listing 5.15: Handler for just heads.

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

Listing 5.16: Do experiment with just heads.

```
term h ≔ just-heads coin.

[] ; [] ‖ do experiment with just-heads
↠ (Do)
[h] ; [finish(h)] ‖ (coin#flip •) ∧ (coin#flip •)
↠ (Perform ×2)
[h] ; [(coin, flip, •, a2),(coin, flip, •, a1),finish(h)] ‖ a2 ∧ a1
↠ (Handle-Value)
[unvalued(h)] ; [(coin, flip, •, a2),(coin, flip, •, a1),finish(h)] ‖ a2 ∧ a1
```

```
↠ (Handle-Performance ×2)
[unvalued(h)] ; [finish(h)] ‖ true ∧ true
↠ (Simplify)
[unvalued(h)] ; [finish(h)] ‖ true
↠ (Handle-Final)
[] ; [] ‖ true
```

Handling alternating possibilities

A more sophisticated handler.

Listing 5.17: Handler for alternating between heads and tails

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

Listing 5.18: Do experiment with alternating between heads and tails.

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

[] ; [] ‖ do experiment with h
↠ (Do)
[h] ; [finish(h)] ‖ (coin#flip •) ∧ (coin#flip •)
↠ (Perform ×2)
[h] ; [(coin, flip, •, a2),(coin, flip, •, a1), finish(h)] ‖ a2 ∧ a1
↠ (Handle-Value)
[unvalued(h)] ; [(coin, flip, •, a2),(coin, flip, •, a1), finish(h)] ‖ (b1 ⇒ a2
    ∧ a1)
↠ (Handle-Performance)
[unvalued(h)] ; [(coin,flip,•,a1), finish(h)]
  ‖ (_ k ⇒ (b2 ⇒ k b2 (not b2))) • (a2 b1 ⇒ a2 ∧ a1)
↠ (Simplify)
[unvalued(h)] ; [(coin,flip,•,a1), finish(h)] ‖ b2 ⇒ b2 ∧ a1
↠ (Handle-Performance)
[unvalued(h)] ; [finish(h)]
  ‖ (_ k ⇒ b3 ⇒ k b3 (not b3)) • (a1 b2 ⇒ b2 ∧ a1)
↠ (Simplify)
```

```
[unvalued(h)] ; [finish(h)] ‖ b3 ⇒ (not b3) ∧ b3
↠ (Handle-Final)
[] ; [] ‖ (f ⇒ f true) (b3 ⇒ (not b3) ∧ b3)
↠ (Simplify)
[] ; [] ‖ false ∧ true
↠ (Simplify)
[] ; [] ‖ false
```

5.3.2 Example: Exception

Resource

Listing 5.19: Resource for exception.

```
type Exceptional (α : Type) : Resource
  ≔ resource{ throw : unit → α }.

term exception (α : Type) : Exceptional α.
  ≔ new (Exceptional α).
```

Experiment

[TODO] think of some experiment

Listing 5.20: Experiments with exceptions.

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

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

Handling exceptions as optionals

Listing 5.21: Handler of exceptions as optionals.

```
term optionalization (α : Type) : Exceptional α -> α ↘ optional α
  ≔ handler
      { #throw • _ ⇒ none
      ; value a    ⇒ some a
      ; final x    ⇒ x }.
```

Listing 5.22: Do division safely with optionalization

```
h ≔ optionalization

[] ; [] ‖ do divide-safely 5 0 with optionalization
↠ (Do)
[h] ; [finish(h)] ‖ divide-safely 5 0
↠ (Simplify)
[h] ; [finish(h)] ‖ exception#throw •
↠ (Perform)
[h] ; [(exception,throw,•,a), finish(h)] ‖ a
↠ (Handle-Value)
[unvalued(h)] ; [(exception,throw,•,a), finish(h)] ‖ some a
↠ (Handle-Performance)
[unvalued(h)] ; [finish(h)] ‖ (x k => none) • (a => a)
↠ (Simplify)
[unvalued(h)] ; [finish(h)] ‖ none
↠ (Handle-Final)
[] ; [] ‖ none
```

5.3.3 Example: Mutability

Implementing the mutable effect using algebraic effect handlers.

Resource

Listing 5.23: Resource for mutability.

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

term total   ≔ new (Mutable int).
term switch  ≔ new (Mutable boolean).
term message ≔ new (Mutable string).
```

Experiment

Listing 5.24: Experiment with mutability.

```
term tally • : unit
  ≔ total#set (1 + (total#get •))

term flipped • : boolean
  ≔ switch#set (negate (switch#get •)) ≫ switch#get •.

term say (s : string) : unit
  ≔ message#set s.
```

Handling

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

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

Listing 5.25: Handler for mutability.

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

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

5.4 Considerations for Algebraic Effect Handlers

[TODO] Advantages:

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

[TODO] Disadvantages:

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

Chapter 6

Freer-Monadic Effects

code here based on code in `~/Documents/Prototypes/haskell/freer-monads/src`

6.1 The Problem of Composing Monads

Consideration of the problem of *composing monads*; no systems so far have facilitated this while maintaining type-checking

6.2 Freer Monad

[TODO] describe goals:

1. preserve typing rules
2. don't require redundant implementations of same effect-style for each instance
3. separate performances and handlers

6.2.1 Free Monad

[TODO] use standardized type-class instance parameter notation, defined in monadic-effects chapter

Listing 6.1: free monad

```
type Free (ϕ : Type → Type) (α : Type) : Type
  ≔ α + ϕ (Free ϕ α)

term pure (ϕ α : Type) : α → Free ϕ α ≔ left.
term impure (ϕ α : Type) : ϕ (Free ϕ α) → Free (ϕ α) ≔ right.
```

```
term lift-Free (ϕ α : Type)
  : {Functor ϕ} → ϕ α → Free ϕ α
  := impure ∘ map pure.
```

```
instance Functor ϕ ⇒ Monad (Free ϕ)
  { map f fm := match fm
                { pure a ⇒ pure (f a)
                ; impure m ⇒ impure (map (map f) m) }
  ; return a := pure a
  ; fm >>= k := match fm
                  { pure a ⇒ k a
                  ; impure m ⇒ impure (map (m' ⇒ m' >>= k) m) } }.
```

Example: Free Mutable

```
type Free-Mutable (σ : Type) : Type -> Type := Free (Mutable σ).

term get-Free (σ : Type) : Free-Mutable σ σ
  := lift-Free get.

term set-Free (σ : Type) : Free-Mutable σ unit
  := lift-Free set.

// TODO: did I define run-Mutable in monadic-effects?
term run-Free-Mutable
      (m : Free-Mutable σ α) (s : σ)
  : σ × α
  := match m
      { pure a ⇒ (a, s)
      ; impure m' ⇒ let (m'', s') := run-Mutable m' s in
                    run-Free-Mutable m'' s' }.
```

6.2.2 Freer Monad

Left Kan Extension

[TODO] Intuition for freer monad using left Kan extension.

```
type LAN (γ : Type -> Type) (α χ : Type) : Type
  ≔ (γ α) × (χ -> α) -> LAN γ α.

term make-LAN (γ : Type -> Type) (α χ : Type)
              (gx : γ α) (h : χ -> α)
  : LAN γ α χ
  ≔ (gx, h).

instance Functor (LAN γ)
  { map f (make-LAN gx h) ⇒ make-LAN gx (f ∘ h) }.

term lift-LAN (γ : Type -> Type) (α : Type) : γ α -> LAN γ α
  ≔ (h gx ⇒ make-LAN gx h) (x ⇒ x).
```

Definition of Freer Monad

[TODO] Define freer monads.

[TODO] Explanation of how freer monads can model effects

1. left Kan extension (Lan)
2. act as a sort of monadic implementation of algebraic effect handlers

```
type Freer (γ : Type -> Type) (α χ : Type) : Type
  ≔ α + (γ χ -> (χ -> Freer γ χ) -> Freer γ α).

term lift-freer (u : γ α)
  : Freer γ α
  ≔ impure u pure.

// TODO: move this somewhere else.. to monad section I guess
term fish f g a = f a >>= g
notation "f >=> g" ::= "fish f g"

instance (γ : Type) => Monad (Freer γ)
  { map f fm ⇒ match fm
               { pure a ⇒ pure (f a)
               ; impure u q ⇒ impure u (map f ∘ q) }
  ; return a ⇒ pure a
```

```
  ; fm >>= k ⇒ match fm
                 { pure a ⇒ k a
                 ; impure u k' ⇒ impure u (k' >=> k) } }.
```

6.3 Poly-Freer Monad

[TODO] Explanation of adding a stack of freer monadic effects to large container, which yields another freer monad.

[TODO] Basically what Haskell' polysemy implements, but here will do it more simply

[TODO] Need to introduce dependent types? Or at least briefly explain

Listing 6.2: Definition of Poly-Freer Monad

```
type Poly-Freer (L : list Type) (α : Type) : Type
  := (M : Type -> Type) => Monad m ->
    (χ => Type-List-Contains σ M -> M χ) ->
    M α

instance Monad (Poly-Freer σ) ...
```

6.4 Freer-Monadic Effects

[TODO] Demonstration of stateful freer monad, paralleling monadic example from chapter 4.

6.5 Discussion

[TODO] Discussion of advantages and disadvantages of freer monadic effects

1. fully-typed system for algebraic effect handlers; all the advantages of algebraic effect handlers
2. facilitates generally composable effects
3. not implementable in non-dependently typed languages like Haskell, OCaml, SML, etc. since it requires extensive type-level manipulation
4. potentially very user-unfriendly, incurring many of the original problems with monadic effects

Appendix A

Appendix: Prelude for 𝔸

A.1 Functions

```
term compose (α β γ : Type)
  : (α -> β) -> (β -> γ) -> (α -> γ)
  ≔ f g a ⇒ f (g a).
```

A.2 Data

Listing A.1: unit

```
primitive type unit : type.
```

Listing A.2: boolean

```
type boolean : Type : unit ⊕ unit.
```

Listing A.3: natural number

```
type natural : Type ≔ unit ⊕ natural.

term zero : natural ≔ left •.
term succ : natural -> natural ≔ right
```

```
term (+) (m n : natural) : natural
  ≔ case m
      { zero    => n
      ; succ m' => succ (m' + n) }.

term (*) (m n : natural) : natural
  ≔ case m
      { zero    => zero
      ; succ m' => n + m' * n }.
```

Listing A.4: integer

```
// left is negative, right is positive
type integer : Type ≔ natural ⊕ natural.

term 0 : integer ≔ right zero.
term 1 : integer ≔ right (succ zero)

//TODO: define (+)

term (-) (i : integer) : integer
  ≔ case i
      { left  n => right n
      ; right n => left  n }.

term (-) (i j : integer) : integer ≔ i + (-j)
```

Listing A.5: rational

```
//TODO
```

A.3 Data Structures

Listing A.6: optional

```
type optional (α : Type) : Type ≔ α ⊕ unit.

term some (α : Type) (a : α) : optional α ≔ left a.
```

```
term none (α : Type) : optional α ≔ right •.
```

Listing A.7: list

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

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

Listing A.8: list utilities

```
// undefined on nil
term head (α : Type) (as : list α) : α
  ≔ case as{ a ∷ _ ⇒ a }.

term tail (α : Type) (as : list α) : list α
  ≔ case as{ _ ∷ as' ⇒ as' }.

term concat (α : Type) (as1 as2 : list α) : list α
  ≔ case as
      { []        ⇒ as2
      ; a ∷ as1' ⇒ a ∷ concat as1' as2 }.

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

instance Functor list :=
  { map a f as
      ≔ case as
          { []      ⇒ []
          ; a ∷ as' ⇒ f a ∷ map f as' } }.
```

Listing A.9: list notations

```
[]   ::=   nil

«term»1 ∷ «term»2   ::=   app «term»1 «term»2
```

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

Listing A.10: tree and forest

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

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

Listing A.11: mapping

```
type mapping (α β : Type) : Type := α -> β.

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

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

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

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

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


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


\section{Classes}


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

Listing A.12: Equal

```
type (=) : Type -> Type -> Type := a => a = a

class (=) (α β : Type)
  { reflex : α => α = α }

primitive term reflex (α : Type) : α = α.
primitive term
```