Chapter 1

Introduction

1.1 Outline

1. Definition of a simple, typed lambda calculus
2. Context and definition of *effects* in programming languages
3. Examples of common effects
   (a) IO
   (b) mutable data, state
   (c) exception
   (d) nondeterminism
   (e) non-termination / non-totality
   (f) reader, writer, output
   (g) continuation
4. Explanation of how effects are handled in ML's style
   (a) call-by-value at runtime
   (b) examples of effects: IO, exceptions
   (c) effects are untyped
   (d) simple exception-handling system (type/catch)
5. Explain why this style is not purely functional
   (a) exposes *side-effects*, which are implicit effects not accessible by the programming language

1.2 A Simply Typed Lambda-Calculus

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 $\mathbb{\Lambda}$.

1.2.1 Definition of $\mathbb{\Lambda}$

TODO: include let bindings for $\mathbb{\Lambda}$?

Syntax for $\mathbb{\Lambda}$

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

metavariable	constructors	name
$\langle term \rangle$	$\langle term\text{-}name \rangle$	atom
	$\langle name \rangle : \langle type \rangle \Rightarrow \langle term \rangle$	function
	$\langle term \rangle\ \langle term \rangle$	application
	`let` $\langle name \rangle \coloneqq \langle term \rangle$ `in` $\langle term \rangle$	bind
	`left` $\langle term \rangle$	left
	`right` $\langle term \rangle$	right
	`case` $\langle term \rangle$ { `left` $\langle name \rangle \Rightarrow \langle term \rangle \mid$ `right` $\langle name \rangle \Rightarrow \langle term \rangle$ }	case
	$(\langle term \rangle, \langle term \rangle)$	product
	`first` $\langle term \rangle$	first
	`second` $\langle term \rangle$	second
$\langle type \rangle$	$\langle type\text{-}name \rangle$	atom
	$\langle type \rangle \to \langle type \rangle$	arrow
	$\langle type \rangle + \langle type \rangle$	sum
	$\langle type \rangle \times \langle type \rangle$	product

(1.1)

where $\langle term\text{-}name \rangle$ and $\langle type\text{-}name \rangle$ are metavariables that range over an infinite collection of distinct names, and $\langle name \rangle$ ranges over all names. A given expression is well-formed with respect to $\mathbb{\Lambda}$ if it is constructable by a series of applications of these rules. Note that this syntax does not specify any concrete terms or types. $\mathbb{\Lambda}$'s syntax is defined abstractly in order for the definition its typing and semantics as simple as possible. For the sake of intuition however, these are some examples of the likes of which these meta-variables stand in for:

- $\langle term\text{-}name \rangle$: $0, 1, 2$, `true`, `false`, `"hello world"`, $\bullet$.
- $\langle type\text{-}name \rangle$: `natural`, `integer`, `boolean`, `string`, `void`, `unit`.

- $\langle name \rangle$: `x`, `y`, `this-is-a-constant`

TODO: include some function terms and (and maybe types?)

Typing Rules for $\mathbb{A}\!\!\backslash$

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 1.1, 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

$$\frac{P_1 \qquad \cdots \qquad P_n}{Q},$$

which asserts that the premises $P_1, \ldots, P_n$ entail the conclusion Q. For example,

$$\frac{\Gamma \text{ is a judgement context} \qquad a \text{ is a term} \qquad \alpha \text{ is a type} \qquad \Gamma \vdash a : \alpha}{\Gamma \vdash a : \alpha}$$

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 con texts,
- $a, b, c, \ldots$ each range over terms,
- $\alpha, \beta, \gamma, \ldots$ each range over types.

The following typing rules are given for A\:

$$\textbf{TODO} \quad \Gamma, a : \alpha \;\vdash\; a : \alpha$$

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

$$\textsc{Function-Concretization}\quad \frac{\Gamma \;\vdash\; a : \alpha \to \beta \qquad \Gamma \;\vdash\; b : \alpha}{\Gamma \;\vdash\; a\, b : \beta}$$

$$\textsc{Construct-Sum-Left}\quad \frac{\Gamma \;\vdash\; a : \alpha}{\Gamma \;\vdash\; (\texttt{left}\ a) : \alpha + \beta}$$

$$\textsc{Construct-Sum-Right}\quad \frac{\Gamma \;\vdash\; b : \beta}{\Gamma \;\vdash\; (\texttt{right}\ b) : \alpha + \beta}$$

$$\textsc{Construct-Product}\quad \frac{\Gamma \;\vdash\; a : \alpha \qquad \Gamma \;\vdash\; b : \beta}{\Gamma \;\vdash\; (a, b) : \alpha \times \beta}$$

$$\textsc{Destruct-Sum}\quad \frac{\Gamma \;\vdash\; x : (\alpha + \beta) \qquad \Gamma, a : \alpha \;\vdash\; c_1 : \gamma \qquad \Gamma, b : \beta \;\vdash\; c_2 : \gamma}{\Gamma \;\vdash\; (\texttt{case}\ x\ \{\ \texttt{left}\ a \Rightarrow c_1 \ \mid\ \texttt{right}\ b \Rightarrow c_2\ \}) : \gamma}$$

$$\textsc{Destruct-Product-First}\quad \frac{\Gamma \;\vdash\; x : (\alpha \times \beta)}{\Gamma \;\vdash\; (\texttt{first}\ x) : \alpha} \tag{1.2}$$

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\:

$$\beta\textsc{-Reduce}\quad \frac{\Gamma \;\vdash\; (a : \alpha \Rightarrow b) : (\alpha \to \beta) \qquad \Gamma \;\vdash\; v : \alpha \qquad \mathsf{value}\ v}{(a : \alpha \Rightarrow b)\ v \twoheadrightarrow [v/a]b} \tag{1.3}$$

The most fundamental of these rules is 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 comptuation.

1.2.2 Other Structures in $\mathbb{A}\backslash$

TODO: decide, based on future developments, which terms and types should be included here. Things like:

- integer
- natural
- boolean
- unit

1.2.3 Properties of $\mathbb{A}\backslash$

With the syntax, typing rules, and reduction rules for $\mathbb{A}\backslash$, we now have a completed definition of the language. However, some of the design desicions may seem arbitrary even if intuitive. This particular framework is good because it maintains a few nice properties that make reasoning about $\mathbb{A}\backslash$ intuitive and extendable.
TODOdefine these properties in English; want to keep prog-language and meta-language separate.

Theorem 1.2.1. *(Type-Preserving Substitution in $\mathbb{A}\backslash$).* If $\Gamma, a : \alpha \ \vdash \ b : \beta$, $\Gamma \ \vdash \ v : \alpha$, and $\mathsf{value}\ v$, then $\Gamma \ \vdash \ [v/a]b : \beta$.

Theorem 1.2.2. *(Reduction Progress in $\mathbb{A}\backslash$).* If $\{\} \ \vdash \ a : \alpha$, then either $\mathsf{value}\ a$ or $\exists a' : a \twoheadrightarrow a'$.

Theorem 1.2.3. *(Type Soundness in $\mathbb{A}\backslash$).* If $\Gamma \ \vdash \ a : \alpha$ and $a \twoheadrightarrow a'$, then either $\mathsf{value}\ a'$ or $\exists a'' : \alpha \twoheadrightarrow a''$.

Theorem 1.2.4. *(Type Preservation in $\mathbb{A}\backslash$).* If $\Gamma \ \vdash \ a : \alpha$ and $a \twoheadrightarrow a'$, then $\Gamma \ \vdash \ a' : \alpha$.

Theorem 1.2.5. *(Strong Normalization in $\mathbb{A}\backslash$).* For any term a, either $\mathsf{value}\ a$ or there is a sequence of reductions that ends in a term a' such that $\mathsf{value}\ a$.

1.3 Computation with Effects

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

The definition of $\mathbb{A}\backslash$ in section 1.2.1 embodies a good flavor for many similar functional 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*.

1.3.1 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 1.3.1. 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 cababilities depend on factors outside of their usual scope. The only

[1]**TODO**: source

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 1.3.2. 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 1.3.3. A program is *impure* if it has effects.

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.
- *mutable data*, e.g. mutable variables, in-place arrays.
- *exception*, e.g. division by 0, out-of-bounds index of array.
- *nondeterminism*, e.g. **TODO**: what would be good examples for this...
- *partiality*, e.g. **TODO**
- *continuation*, e.g. **TODO**

TODO: go into detail here, or till after I've talked about comparing functional/imperative approaches to effects?

According to definition 1.3.1, 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. Overally languages have clustered into two groups.

- *Functional* 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 functional 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).

1.4 A Simple Approach to Effects

??

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 effect design strategy will be demonstrated using $\mathbb{B}$. $\mathbb{B}$ extends $\mathbb{A}$ with the following new features:

- *Sequence.* A collection of terms is arranged in the sequence of desired resolution. Reducing the sequence term will reduce the terms in order.

- *Mutable.* A new type that indicates mutable data. It must be explicitly read from and wrote to, rather than handled like the normal (immutable) value.

- *Exception.* A collection of new term constructs, allowing for programs to *throw* and *catch* exceptions that interrupt usual reduction.
- *Input/Ouput (IO)* A collection of new primitive terms that, when evaluated, yield effects in the evaluation context not reflected in their types.

1.4.1 Syntax for $\mathbb{B}$

metavariable	constructors	name
$\langle term \rangle$	$\langle term \rangle$; $\langle term \rangle$	sequence
	$*\langle term \rangle$	mutable
	$!\langle term \rangle$	read
	$\langle term\text{-}name \rangle \leftarrow \langle term \rangle$	write
	`throw` $\langle exception\text{-}name \rangle(\langle term \rangle)$	throw
	`catch` $\{ \langle exception\text{-}name \rangle(\langle name \rangle) \Rightarrow \langle term \rangle \}$ `in` $\langle term \rangle$	catch
$\langle type \rangle$	`mutable` $\langle type \rangle$	mutable

(1.4)

Additionally, the following new primitive terms are introduced to provide IO effects.

$$\begin{aligned} \texttt{input} &: \texttt{unit} \to \texttt{string} \\ \texttt{output} &: \texttt{string} \to \texttt{unit} \end{aligned} \tag{1.5}$$

1.4.2 Typing Rules for $\mathbb{B}$

$$\textsc{Construct-Sequence} \quad \frac{\Gamma \vdash a : \alpha \qquad \Gamma \vdash b : \beta}{\Gamma \vdash (a \;;\; b) : \beta}$$

$$\textsc{Construct-Mutable} \quad \frac{\Gamma \vdash a : \alpha}{\Gamma \vdash (*a) : \texttt{mutable}\ \alpha}$$

$$\textsc{Construct-Mutable-Read} \quad \frac{\Gamma \vdash m : \texttt{mutable}\ \alpha}{\Gamma \vdash (!m) : \alpha}$$

$$\textsc{Construct-Mutable-Write} \quad \frac{\Gamma \vdash m : \texttt{mutable}\ \alpha \qquad \Gamma \vdash a : \alpha}{\Gamma \vdash (m \leftarrow a) : \texttt{unit}}$$

$$\textsc{Construct-Throw} \quad \frac{\textsf{exception}\ e\ \textsf{of}\ \xi \qquad \Gamma \vdash x : \xi}{\Gamma \vdash (\texttt{throw}\ e(x)) : \alpha}$$

$$\textsc{Construct-Catch} \quad \frac{\textsf{exception}\ e\ \textsf{of}\ \xi \qquad \Gamma, x : \xi \vdash a_1 : \beta \qquad \Gamma \vdash a_0 : \alpha}{(\texttt{catch}\ \{\ e(x) \Rightarrow a_1\ \}\ \texttt{in}\ a_0) : \alpha} \tag{1.6}$$

1.4.3 Reduction Rules for $\mathbb{B}$

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: formally explain how mutable environments work (M). Models the storing of values for mutables.

Reduction Rules for Sequences

$$\begin{gathered}
\textsc{Sequence-Reduce}\quad \frac{\mathcal{S}, \mathcal{E} \parallel a \;\twoheadrightarrow\; \mathcal{S}', \mathcal{E}' \parallel a'}{\mathcal{S}, \mathcal{E} \parallel (a\ ;\ b) \;\twoheadrightarrow\; \mathcal{S}', \mathcal{E}' \parallel (a'\ ;\ b)} \\[1em]
\textsc{Executed}\quad \frac{\mathsf{value}\ a}{\mathcal{S}, \mathcal{E} \parallel (a\ ;\ b) \;\twoheadrightarrow\; \mathcal{S}, \mathcal{E} \parallel b}
\end{gathered} \tag{1.7}$$

Reduction Rules for Mutables

$$\begin{gathered}
\textsc{Initialize}\quad \frac{\mathsf{value}\ v}{\mathcal{S} \parallel\ * v \;\twoheadrightarrow\; \mathcal{S}\llbracket \mathbb{i} \mapsto v \rrbracket \parallel \mathbb{i}} \quad \text{where } \mathbb{i} = \mathsf{new\text{-}uid}(\mathcal{S}) \\[1em]
\textsc{Mutable-Reduce}\quad \frac{a \;\twoheadrightarrow\; a'}{*a \;\twoheadrightarrow\; * a'} \\[1em]
\textsc{Read}\quad \mathcal{S}\llbracket \mathbb{i} \mapsto v \rrbracket \parallel\ !\mathbb{i} \;\twoheadrightarrow\; \mathcal{S}\llbracket \mathbb{i} \mapsto v \rrbracket \parallel v \\[1em]
\textsc{Read-Reduce}\quad \frac{m \;\twoheadrightarrow\; m'}{m \;\twoheadrightarrow\; !m'} \\[1em]
\textsc{Write}\quad \frac{\mathsf{value}\ v'}{\mathcal{S}\llbracket \mathbb{i} \mapsto v \rrbracket \parallel \mathbb{i} \leftarrow v' \;\twoheadrightarrow\; \mathcal{S}\llbracket \mathbb{i} \mapsto v' \rrbracket \parallel \bullet} \\[1em]
\textsc{Write-Source-Reduce}\quad \frac{m \;\twoheadrightarrow\; m'}{m \leftarrow a \;\twoheadrightarrow\; m' \leftarrow a} \\[1em]
\textsc{Write-Target-Reduce}\quad \frac{a \;\twoheadrightarrow\; a'}{m \leftarrow a \;\twoheadrightarrow\; m \leftarrow a'}
\end{gathered} \tag{1.8}$$

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).

$$\textsc{Throw-Reduce}\quad \frac{\textsf{exception}\ e\ \textsf{of}\ \xi \qquad a \twoheadrightarrow a'}{\texttt{throw}\ e(a) \twoheadrightarrow \texttt{throw}\ e(a')}$$

$$\begin{aligned}
&\textsc{Throw}\quad \frac{\textsf{exception}\ e\ \textsf{of}\ \xi \qquad \textsf{value}\ v}{\mathcal{E} \parallel \texttt{throw}\ e(v) \twoheadrightarrow \texttt{throw}\ e(v) + \mathcal{E} \parallel \texttt{throw}\ e(v)} \\
&\textsc{Raise}\quad \frac{\textsf{exception}\ e\ \textsf{of}\ \xi}{\texttt{throw}\ e(v) + \mathcal{E} \parallel a \twoheadrightarrow \texttt{throw}\ e(v) + \mathcal{E} \parallel \texttt{throw}\ e(v)}
\end{aligned} \tag{1.9}$$

$$\textsc{Catch}\quad \frac{\textsf{exception}\ e\ \textsf{of}\ \xi \qquad \textsf{value}\ v}{\texttt{throw}\ e(v) + \mathcal{E} \parallel \texttt{catch}\ \{\ e(x) \Rightarrow a_1\ \}\ \texttt{in}\ a_0 \twoheadrightarrow \mathcal{E} \parallel [v/x]a_1}$$

Reduction Rules for IO

$$\begin{aligned}
\textsc{Input}\quad & \mathcal{O} \parallel \texttt{input}\ \bullet \twoheadrightarrow \mathcal{O} \parallel \mathcal{O}(\texttt{input}\ \bullet) \\
\textsc{Output}\quad & \mathcal{O} \parallel \texttt{output}\ s \twoheadrightarrow \mathcal{O}(\texttt{output}\ s) \parallel \bullet
\end{aligned} \tag{1.10}$$

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 $\mathbb{B}$'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 $\mathbb{B}$ must satisfy the following specifications:

Specification of $\mathcal{O}$

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

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

Listing 1.1: Hello World

```
output "hello world"
```

But as far as the definition of $\mathbb{B}$ is concerned, this term is treated just like any other term that evaluates to $\bullet$. 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` $\bullet$):

1. Prompt the console for user text input.
2. Interpret the user text input as a string, then return the string.

➢ $\mathcal{O}$(`output` s):

1. Write s to the console.
2. Resolve to $\mathcal{O}$.

As intended by $\mathbb{B}$'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 $\mathcal{O}$ however, $\mathbb{B}$ does not guarantee anything about how $\mathcal{O}$ behaves. Consider, for example, this alternative implementation:

Implementation 2 of $\mathcal{O}$:

➢ $\mathcal{O}$(`input` $\bullet$):

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

➢ $\mathcal{O}$(`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 $\mathcal{O}$.

This implementation does not seem to reflect the design of $\mathbb{B}$, though unfortunately it is still compatible. In the way that $\mathbb{B}$ 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 $\mathcal{O}$.

1.5 Motivations

This chapter has introduced the concept of effects in programming languages, and presented $\mathbb{B}$ as a simple approach to extending a simple lambda calculus, $\mathbb{A}$, 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 2

Monadic Effects

2.1 Outline

1. Definition and context for monads in category theory and computer science
2. Explanation of how monads can model effects in general
3. Demonstration of constructing and using the stateful monad, building it up from scratch in Haskell or Haskell-like psuedocode
   (a) Explain the Functor, Applicative, and Monad typeclasses, and how they build up the mathematical definition of a monad
4. Outline some problems with monadic effects
   (a) different effects are not composable
   (b) paper: *The Awkward Squad*

2.2 Introduction to Monads

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.

To intuit the monadic style for modeling implicit contexts, first consider the *state* effect. In section **??**, the language $\mathbb{B}$ presented an implementation of mutable data through several new syntactic structures and an evaluation-time mutable data context. However, there is a way to write programs that implement similar sorts of features within just $\mathbb{A}$ by making stateful computations explicit. The style is to explicitly pass state to stateful computations as a parameter. In this way, the type of a computation with state σ that returns α is

$$\texttt{state}\ \sigma\ \alpha := \sigma \to (\sigma \times \alpha). \tag{2.1}$$

This type represents how a stateful computation takes as input an initial state, and returns as output the modified state as well as the result of the computation. In other words, a term of type `state` $\sigma\ \alpha$, when given an initial state of type σ, is evaluated to a term of type α and the resulting state modified by the computation. This style has the advantage of being pure, and the disadvantage of making explicit what is conceptually better kept implicit.

We can provide implementations of `get` and `set` that model the same mutable state effect as in $\mathbb{B}$, *without* positing them as language primitives with special typing and reduction rules. They are defined as follows:

$$\texttt{read} := s : \sigma \Rightarrow (s, s) \ : \ \texttt{state}\ \sigma\ \sigma \tag{2.2}$$

$$\texttt{write} := s' : \sigma \Rightarrow (s : \sigma \Rightarrow (s', \bullet)) \ : \ \sigma \to \texttt{state}\ \sigma\ \texttt{unit} \tag{2.3}$$

Note that we have not needed to require *truely* mutable data to be available. This setup may seem limited, in that only "one state," α, is kept track of as opposed to the table of mutables made available by $\mathbb{B}$. However α, as an arbitrary type, can in fact store multiple values in the same way as in $\mathbb{B}$, for example by using product types. If $\alpha = \alpha_1 \times \cdots \times \alpha_n$, then each α_i can correspond to one mutable store in a $\mathbb{B}$ state context.

So what does this kind of stateful computation looks like in action? Consider the following term that adds 1 to its integer state:

```
increment : state int • :=
  s ⇒ write (s + 1)
```

Let us try to make our tools more general. To update the state with an arbitrary function:

```
modify : (σ → σ) → state σ • :=
  f s ⇒ write (f s)
```

With this, we can easily rewrite the `increment` as

```
increment : state int • :=
  modify (x ⇒ x + 1)
```

How does this expression of stateful computation look in sequence?

```
sequence : state σ α → state β α → state σ β :=
  sa sb s0 ⇒
    let (s1, a) = sa s0 in
    sb s1
```

But what if we wanted to allow g to use the output of f? We can invent a new kind of sequence, a *binding sequence*. The following term defines a *stateful bind*.

```
bind : state σ α → (α → state σ β) → state σ β :=
  sa sf s0 ⇒
    let (s1, a) = sa s0 in
    sf a s1
```

This framework allows The mutability is pushed to the higher layer of abstraction or passing along the state with each function, rather than having it be an interface external to the