Chapter 1

A Simple Approach to Effects

[TODO] add captions to all listings

1.1 Declarative and Imperative Programming Languages

Previously in section ??, we defined 𝔸 as a basic variant of the λ-calculus. In a larger context of programming languages in general, 𝔸 falls under the definition of a declarative programming language and gives a good flavor for the class. **Declarative** programming languages have a style focussed on *mathematically defining how computations are defined* i.e. the logic rather that execution flow. **Imperative** programming languages, in contrast, have a style focussed on *instructing how computations are carried out* i.e. the execution flow that logic rather. The differences between these kinds arise only in high-level programming languages, because low-level languages (assembly and C-level languages) are by definition instructing the computer what to do roughly step-by-step. Additionally, many languages implement features from both of these categories.

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

> **(Declaratively) Sum of a list of integers**. Let l be a list of integers. If l is empty, then the sum of l is 0. Otherwise, l is not empty, so let h, t be the head and tail of l respectively. Then the sum of l is h plus the sum of t.

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

> **(Imperatively) Sum of a list of integers**. Let l be a list of integers. Declare s to be an integer variable initialized to 0. For each i from 1 to the length of l, let x be the i-th element of l and set s to $s + x$. Then the sum of l is the final s.

This example demonstrates the conceptual difference between the two styles. In the declarative style, the algorithm describes what the sum of a list of integers in terms of an inductive understanding of a list as either empty or a head integer and a tail list. This naturally yields a recursive definition to match an inductive list. On the other hand, in the

imperative style, the algorithm describes how to keep track of the partial sum while stepping through the list. This naturally yields a definition with a while loop and a variable[1] to store the partial sum.

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

- **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.

1.2 Computation with Effects

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

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

```
TODO
```

In imperative languages, it is rarely the case that evaluation is context-free. As a simple example, just consider mutability. This Java program defines a class with two functions that each mutate its internal variable.

```
class A {
  int x;

  public A(int x) {
```

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

```
    this.x = x;
  }

  public void increment() {
    this.x = this.x + 1;
  }

  public void triple() {
    this.x = this.x * 2;
  }
}
```

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

```
public int f1() {
  A a = new A(0);
  a.increment();
  a.triple();
  return a.x;
}

public int f1() {
  A a = new A(0);
  a.triple();
  a.increment();
  return a.x;
}
```

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

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

compile to assembly code, do not allow reference to specific registers or memory pointers, so in higher-level languages they are among the implicit contexts.

In the early days of computer engineering, often a program was written to be run by a single, unique physical machine. Still today, a program is written as a code to be run on a physical machine[2]. The unique aspects of that machine still have an impact on how that running actually happens of course, but the development has been to increase the level of abstraction of the program, for example so that a program can be written in as a code to be run on many very different computers. This first level of abstraction — the step from working on one computer to working on many computers — is another example of a programming language making a context implicit; abstracting in this way makes the specific computer that the program's code is running on a part of the implicit context.

However practically beneficial this abstraction using implicit contexts is, it does not come for free. In the definition of 𝔸, there appear to be no implicit contexts at all — every term can be fully evaluated with only rules defined by the language[3]. In fact, 𝔸 is defined such that the execution of its programs is completely independent from the physical state of the world. This is very useful for formal analyses, since 𝔸 programs exist purely as mathematical structures. However, this aspect also yields a language that is inert relative to the physical world; 𝔸 cannot *do* anything that requires reference to an implicit context. For example, it is definitionally impossible to write the standard "hello world"[4]. program in 𝔸, since printing to the screen requires the implicit context of all the mechanisms involved in the action of making a few pixels change color. As an be seen from this lacking, there are a lot of things that people, including professional programmers, would like to be able to do in a language that something as purely-mathematical as 𝔸 is unable to facilitate.

> [**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).

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

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

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

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

programming languages are imperative: Java, C, Python, C++, C#, .NET, JavaScript, PHP, SQL, Go, R[5]. So why has this work introduced 𝔸 in the first place?

From however unpopular origins, the benefits of some features from more purely-mathematical, declarative languages have started making their way into popular programming language design. These adoptions have primarily taken two forms: (1) Convenient semantic structures, such as *λ-expressions* (a.k.a. *anonymous functions*) which were introduced into standard Java, JavaScript, and C++ in the early 2010's. (2) Abstraction-friendly and safer type systems, such as *generics* which were introduced into Java and C++[6] in the early 2000's, and the development of Scala, a functional language that compiles to Java virtual machine byte code. It turns out that having more easily-analyzable code is very useful for building effective, large-scale software. There is a lot to discuss in this direction, especially the topic of formal specification and verification, but that is beyond the scope of this work.

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

The Dilemma of Implicit Contexts

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

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

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

1.2.1 Effects

Effects are the interfaces to implicit contexts that exist syntactically as well as semantically in programming languages. In this way, implicit contexts can also be embedded among syntactical contexts (e.g. scopes) as well as non-syntactical contexts (e.g. execution state);

[5] *10 Most Popular Programming Languages In 2020: Learn To Code.* https://fossbytes.com/most-popular-programming-languages/

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

what is *implicit* from the point of view of a piece of code depends both on its context in the program in which its embedded and its context in the execution state in which it's evaluated. For a given expression with a given scope, the *Explicit* factors (i.e. factors that can be referenced from within the expression) that affect the expression's evaluation are called *in-scope*, and the *implicit* factors (i.e. factors that cannot be referenced from within the expression) that affect the expression's evaluation are called *out-of-scope*.

Definition 1.2.1. For an expression, a computational **effect** is a factor that affects the expression's evaluation but is outside the expression's normal scope.

An expression's *normal scope* is defined by the expression's used language, where "normal" indicates that some languages may have exceptional, abnormal scope-rules for certain circumstances that still yield effects. For example, many languages have *built-in* functions that, while appearing to be normal functions written in the language, actually directly interface with lower-level code not expressible in the language. Some of these built-in functions has exceptional rules that allow it to refer to special factors that are indeed outside of a normal expression's scope. For example, a built-in function common to most programming languages is the *print* function, which sends some data to a peripheral output (such as you're computer's screen). Since *print* is an interface to the implicit context of the mechanics behind handling that peripheral, it is effectual.

Definition 1.2.2. An expression is **impure** if it has effects.

There are a variety of effects that are available in almost every programming language. Interfaces to these effects are usually implemented as impure built-in functions (e.g. *print*) that are used just like any pure functions. [7] I shall use four common effects as running examples throughout the rest of this work: *input/ouput*, *mutability*, *exception*, and *nondeterminism*. They are detailed in the following paragraphs.

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

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

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

Exception. This is the effect of an expression yielding an invalid value according to its specification. A language's interface to exception has two components: throwing an exception and catching a possible exception. Throwing an exception indicates the yielding of an invalid value. Catching a possible exception in an expression is the structure of anticipating a throw of the exception during the evaluation of an expression, and having a pre-defined way of responding to such a throw should it occur. Examples of instances where exceptions are thrown: division by `0`, out-of-bounds index of array, head of an empty list.

> [**TODO**] more examples of exception?

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

> [**TODO**] better examples of nondeterminism?

Sequencing. This is the effect of dictating the order in which other effects are performed. Sequencing is a very simple effect and is useful especially in declarative programing languages that aren't structured around the ordering of performances like imperative languages are.

However, many expressions both in declarative and imperative languages do not have effects. For example, suppose we have a function `max` that takes two integers as input and returns that larger one. If `max` is designed and implemented exactly to this specification, then `max` has no effects; the evaluation of `max` given some integers is not influenced by any factors outside of its normal scope, since the result is completely determined by its two explicit integer parameters.[8]

Definition 1.2.3. A program is **pure** if it has no effects.

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

So now the sides of the Dilemma of Implicit Contexts can be considered more specifically given the concept of "effect". As is made clear by definition 1.2.1, there is an easy method for transforming an impure expression into a pure expression: add the implicit factors depended upon by the expression's effects to the program's scope — now they are explicit and thus no longer effects! This reveals a more formal way of reasoning about effects in the terminology of programming language design. Still there is a dilemma of course, so using this intuition we can reformulate the Dilemma of Implicit Contexts in the terminology of effects:

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

The Dilemma of Effectual Purity

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

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

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

1.3 Definition of 𝔹

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

1.3.1 Primitives in 𝔹

Mutability

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

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

Listing 1.1: Primitives for mutablility

```
primitive type mutable : Type → Type.

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

Listing 1.2: Notations for mutability.

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

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

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

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[9].

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

Instances of `exception-of 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 `(exception-of α → α → β)` for handling any `α`-exceptions while evaluating a given term of type `β`. If an exception is thrown while evaluating a `b : β`, then `catching`

[9]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.

results in the continuation, supplied with the specific thrown exception and its argument, instead of continuing to evaluate *b*.

Table 1.1: Exception syntax for IB

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

Listing 1.3: Definitions for exception

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

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

Listing 1.4: Notation for exception

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

::=

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

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

```
1 exception division-by-0 of integer.
2
3 term division (i j) : integer
4   ≔ if j == 0
5       then throw division-by-0 j
6       else (divide as usual)
```

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

Notice how the term `throw division-by-0 j` on line 5 is typed as an `integer`, yet it is not an integer. This is where the impure nature of exceptions is clear. The throwing of an exception depends on the implicit context of some surrounding `catch` structure in order to behave as if `division` is properly typed. In the case that there is no surrounding `catch`, an implicit default catching mechanism will be triggered — usually a language-external error.

Input/Output (IO)

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 `string`s. However, it is easy to imagine other specific IO functions that would work in a similar manner (e.g. `input-integer`, `input-time`, `output-image`, etc.).

Listing 1.5: Definitions for IO

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

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

Nondeterminism

In 𝕀𝔹, nondeterminism is simply implemented via a special interface similar to how IO is implemented. The following functions define that interface to the language.

Listing 1.6: Primitives for nondeterminism

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

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

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

term random-bool : unit → boolean
```

```
  := (x => x < 0.5) ∘ random-float.
```

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 effects are to be performed. Such an expression is an application of the *sequencing* effect. A primitive term allows sequences to be written, and 𝔹's reduction rules mirror them by reducing terms in the correct order.

Listing 1.7: Primitive for sequencing

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

Listing 1.8: Notation for sequencing

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

1.3.2 Reduction Rules for 𝕀𝔹

[**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)

[**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 Mutability

Table 1.2: Reduction in 𝕀𝔹: Mutability

$$\textsc{Initialize}\quad \frac{\text{value } v}{\mathcal{S} \parallel \star v \quad \twoheadrightarrow \quad new(\mathcal{S}, v) \parallel \bullet}$$

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

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

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 1.3: Reduction in IB: Exceptions

$$\textsc{Throw}\quad \frac{\varepsilon : \texttt{Exception}\ \alpha \qquad v : \alpha \qquad \text{value } x}{\mathcal{E} \parallel \texttt{throw}\ \varepsilon\ x \twoheadrightarrow (\varepsilon, x) :: \mathcal{E} \parallel \blacklozenge}$$

$$\textsc{Raise}\quad \mathcal{E} \parallel \texttt{a}\ \blacklozenge \twoheadrightarrow \mathcal{E} \parallel \blacklozenge$$

$$\textsc{Raise}\quad \mathcal{E} \parallel \blacklozenge\ \texttt{b} \twoheadrightarrow \mathcal{E} \parallel \blacklozenge$$

$$\textsc{Catch}\quad \frac{\varepsilon : \texttt{Exception}\ \alpha}{(\varepsilon, x) :: \mathcal{E} \parallel \texttt{catch}\ \{\ \varepsilon\ a \Rightarrow b\ \}\ \texttt{in}\ \blacklozenge \twoheadrightarrow \mathcal{E} \parallel (a \Rightarrow b)\ x}$$

$$\textsc{Catch-Pass}\quad \frac{\varepsilon : \texttt{Exception}\ \alpha \qquad \text{value } v}{(\varepsilon, x) :: \mathcal{E} \parallel \texttt{catch}\ \{\ \varepsilon\ a \Rightarrow b\ \}\ \texttt{in}\ v \twoheadrightarrow \mathcal{E} \parallel v}$$

TODO: more

Reduction Rules for IO

Table 1.4: Reduction in IB: IO

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

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

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 IB'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 IB must satisfy the following specifications:

Specification of $\mathcal{O}$

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

- $\mathcal{O}$(`input` $\bullet$) 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 1.9: Hello World

```
output "hello world"
```

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

Implementation 1 of 𝒪:

- 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.

Reduction Rules for Nondeterminism

Reduction Rules for Sequences

leftoff

1.4 Motivations

This chapter has introduced the concept of effects in programming languages, and presented 𝔹 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)