2019

A HoTT Approach to Computational Effects

Phillip A. Wells
The College of Wooster, pwells19@wooster.edu

A HoTT Approach to Computational Effects

Independent Study Thesis

Presented in Partial Fulfillment of the
Requirements for the Degree Bachelor of Arts in
the Department of Mathematics and Computer
Science at The College of Wooster

by
Phillip A Wells

The College of Wooster
2019

Advised by:

Nathan Fox

Abstract

A computational effect is any mutation of real-world state that occurs as the result of a computation. We develop a model for describing computational effects within homotopy type theory, a branch of mathematics separate from other foundations such as set theory. Such a model allows us to describe programs as total functions over values while preserving information about the effects those programs induce.

Contents

Abstract **iii**

1 Introduction **1**

2 Homotopy Type Theory **7**

2.1 Type and Space 8

2.1.1 Judgments 8

2.1.2 Universes 9

2.1.3 Functions 10

2.1.4 Sums and Products 13

2.1.5 Finite Types 15

2.1.6 Dependent Types 16

2.1.7 Dependent Product Types 17

2.1.8 Dependent Sum Types 17

2.2 Proof and Logic 18

2.2.1 The Curry-Howard Correspondence 18

2.2.2 Identity Types 19

2.2.3 Propositions 19
2.2.4 Predicate Logic 21
2.2.5 The Univalence Axiom 23

3 Computation **27**
3.1 Formal Languages and Automata 28
3.1.1 Regular Languages and Finite Automata 29
3.1.2 Context-Free Languages and Pushdown Automata 31
3.1.3 Turing Machines and Recursively Enumerable Languages 34
3.2 The Sizes of Infinity 37
3.3 Computation in HoTT 41
3.3.1 Turing Machines as Functions 41
3.3.2 Another Approach: Turing Categories 43
3.3.3 Last Thoughts 44

4 The Coq Proof Assistant **47**
4.1 Inductive Types 47
4.2 Basic Proofs 49
4.3 Records 50
4.4 Functions 52
4.5 HoTT in Coq 53

5 Actions and Effects **55**
5.1 Applications 57
5.1.1 Identity Action 58
5.1.2 Interactive Input 59

5.1.3 Exception Handling 60

6 Conclusion **63**

A 3-State Busy Beaver in Coq **65**

Chapter 1

Introduction

A **computational effect** is any mutation of the state of the "real world" that occurs as a byproduct of computation. Very few programs are effect-free (**pure**, as they are often known); even the simple "hello, world" program takes as its objective the writing of information to some standard output. Any input/output action may be considered effect-inducing (and thus contributing to a program being **impure**), not to mention variable assignment, looping, or exception handling. Effects are everywhere, to put it mildly, and the study of effects has been the subject of much research for decades.

Consider the following snippet of C code:

```
1    int a = add(2, 3);
2    printf("%i\n", a);
```

Assuming these lines were written by a competent, well-meaning programmer, we expect that the integer 5 will be computed by `add()`, and then printed to the standard output. But what if the programmer was less than

capable, or even malicious? The first line provides some basic information about the underlying program: `add()` is a function that takes two integers and returns an integer (or at least a value that can be converted to an integer). Of course, the type signature of a C function is rarely adequate to describe the behavior of the entire subroutine, so `add()` is free to do just about anything else it pleases, like modifying memory, accessing secret information, or firing the proverbial missiles.

Functional programming is a paradigm that treats computation as a series of mathematical function applications, rather than as an explicit sequence of statements. Rather than supplying the machine with a list of instructions and an order to complete them in, a functional program expresses a function definition by specifying what the output of a computation ought to be given some input.

Pure, functional programming languages are better-behaved with respect to effects. Haskell, for instance, requires the programmer to make explicit the flow of data in and out of a program [16]. The upside is that it is much harder to write programs that fail mysteriously. The following lines of code define a function that adds two integers together:

```
1 add :: Integer -> Integer -> Integer
2 add x y = x + y
```

The type declaration on line 1 is the same our `add()` subroutine, with the guarantee that this function maps two integers to an integer *and does nothing else*. If we want to, say, print the output, we must pass the result off to another function written for this purpose, annotated with the expectation that the

function performs an I/O action.

Programming in this way preserves the modularity of code but can be burdensome for the user. If, for instance, a program needs to display a stack trace on an error, then the state of the stack must be passed around along with all the data required to perform the requisite computations. The same goes for incrementing a counter or working with large data structures—any information required to persist must be systematically handed off like a relay baton.

Eugenio Moggi's *Notions of Computation and Monads* is perhaps the most famous solution to the problem of safety versus convenience [11]. The paper presents a categorical analysis of programs that include effectful behavior, and the application of Moggi's strong monads to languages such as F# and Haskell has become the defining feature of the so-called "imperative functional programming style" [9].

While an in-depth understanding of monads in functional programming is not necessary for our purposes here, we give a brief synopsis: a monad in the programmatic sense is a collection of two operations, plus a well-defined method for constructing its elements. These operations provide a general method for creating new types out of existing values—a concept we revisit in Chapter 2—as well as the a method for composing monadic functions [16]. Because monads generalize the notion of effectful computation as something that occurs between non-monadic function applications (and therefore between the evaluation of ordinary expressions) they are sometimes referred to as "programmable semicolons." A monad contributes to the structure of a functional program in much the same way that a semicolon contributes to the

structure of an imperative one.

For all their advantages, monads are somewhat unsatisfying from a type theorist's perspective. They prescribe a mechanism for stepping out of the functional environment when necessary, but they fail to capture the essence of computation with respect to the underlying type system. Haskell uses relatively complex structures where dependent types might be better suited. And because the Haskell compiler restricts the number of instances of a type to one, the Haskell compiler must maintain a global instance table, an unfortunate compromise for a language committed to avoiding shared mutable state [3].

Algebraic effects, introduced by Plotkin and Power in 2002, offers a more recent attempt to marry pure functional programming with computational effects. While monads tend towards the unwieldy when there are many effects to step through, algebraic effect handlers were designed precisely with large and hairy programs in mind [2]. The basic idea is to separate effect declaration, in which the context of the effect is described, from effect handling, where the semantic details are ironed out. The process is not dissimilar to that of declaring a function's type before implementing it, with the added complication that effects are typically described as algebraic data types parameterized by values lifted from the underlying program. For example, an exception handler may exhibit different behavior when processing input/output than when doing arithmetic.

Algebraic effects are a bit easier for the functional lay-programmer to adapt to, given their relatively uncomplicated name and design. Like monads, the underlying mathematics is well-developed and understood, and like

monads, algebraic effects divorce the notion of effectful computation from the surrounding type system. Programs utilizing algebraic effects have the form $A \to \epsilon B$, where A and B are types and ϵ is a possible side effect. The benefit of this approach is that it is easy to determine which values are non-effectful (evaluating an integer will never induce an effect; evaluating an I/O handler applied to an integer is another story). Nevertheless, it would be nice to unify a theory of effects with a theory of types.

The type of a program under such a model is the type of functions that map a value of type A to a value of type B *or* to a computational effect. Rather than treat effects as a separate entity that binds to a value, we treat effectful actions as values in and of themselves. As a simple example, consider the type of function called *main*, intended to replace the C function `main()`. It might map a value of type *string* (the source code of the program) to a value of type *string* or some I/O action. If we wish to perform some computation on some input, we could compose *main* with another function. And while this approach comes at the potential cost of deciding what values are effects, it improves the ability of a type checking program to verify the correctness of programs.

In the following chapters, we develop a model of computational effects in homotopy type theory. We choose this branch of mathematics because of its unique treatment of equality and its Univalence Axiom, which identifies the identity of types with the equivalence of types. The axiom provides a formal basis for function extensionality and guarantees that a theory of effects over sets is isomorphism-invariant. These guarantees have the potential to make reasoning about programs easier, especially given that this model of effects aims to capture computation as a type of total functions over values.

Chapter 2

Homotopy Type Theory

Homotopy type theory (HoTT) is an extension of Per Martin-Löf's intensional type theory [10], which itself is a formalization of the theory of types introduced by Bertrand Russell [12]. It is a foundational language for mathematics, separate from the classical set-theoretic foundation readers may be accustomed to, and entirely absent a logical underpinning such as first-order logic. In exchange, homotopy type theory provides a uniquely flexible mathematics. It relates concepts in type theory to those in homotopy theory (which in turn correspond to concepts in higher category theory), so structures like spaces and paths may be studied without the need for a general topology.

The aim of this chapter is to familiarize readers with the basics of HoTT, and to set a standard for the notation used throughout the thesis. The structures of intensional type theory (hereafter referred to simply as *type theory*) are presented alongside their topological analogues in the hopes of developing a visual intuition for HoTT. We forgo a similar treatment of

category theory, though the relationship between type theory, homotopy theory, and category theory is acknowledged in brief at the end of the chapter.

2.1 Type and Space

2.1.1 Judgments

The conceptual basis of homotopy type theory is **type**. We can visualize a type as a topological space consisting of individual points. Without an underlying predicate logic, we are relatively limited in the observations we can make about types; in fact, there are just two basic **judgments**, or logical rules. The first is

$$a : A,$$

which may be read "a is a type of A, or, equivalently, "a is a point of space A."

Every object in homotopy type theory belongs to a type. It is impossible to discuss untyped objects, because a type is defined by its inhabitants (much in the way that a space is defined by its points). Thus, whenever a new object is introduced, we must introduce its type as well. To express that 0 is a natural number, for example, we write $0 : \mathbb{N}$ ("0 is a point of space $\mathbb{N}$").

The second judgment is that of equality. When two types are judgmentally equal we write

$$A \equiv B,$$

meaning "A and B are equal by definition." The same can be said of objects: $a \equiv b : A$ means that a and b are equal as objects of A. A judgmental equality is

introduced by writing

$$A :\equiv B,$$

read "A is defined as B." Any definitions introduced with :≡ are static, so determining whether two types (or two objects) are equivalent is as easy as reducing terms. Visually, judgmental equality corresponds to symmetry. Spaces A and B (or points a and b) are symmetric if they are the same shape, but presented from different angles.

It is important to note that the two judgments $a : A$ and $a \equiv a$ are not propositions. They cannot be proven or disproven; statements such as $-1 : \mathbb{N}$ and $1 \equiv 2 : \mathbb{N}$ are not false, but syntactically invalid. Restrictive as this may seem to mathematicians, computer scientists should feel at home with types. Just as a compiler for some programming language utilizes a type system to ensure its programs are grammatical, the homotopy type theorist can rest easy knowing that the language of the mathematics itself prevents ill-formed statements.

2.1.2 Universes

Types whose inhabitants are also types are called **universes**. When a new type is introduced, we write $A : \mathcal{U}$, indicating that A is a type in some universe $\mathcal{U}$. There are a couple of ways to consider the structure of universes; we shall favor the "Russellian-style" structure of a cumulative hierarchy. This style implies the existence of a cumulative hierarchy of universes

$$\mathcal{U}_0 : \mathcal{U}_1 : \mathcal{U}_2 : \ ...$$

with no self-referential universes, like the universe of all universes $\mathcal{U}_\infty : \mathcal{U}_\infty$ (so as to avoid the classic Russell's paradox—does the type of all types belong to itself?). When the placement of a universe in the hierarchy is not explicitly required, we can omit the subscripts and simply write $\mathcal{U}$. This property of universes, called **typical ambiguity**, makes working with universes much cleaner, if a bit dangerous; abuse of typical ambiguity would allow us to generate paradoxes, such has the aforementioned Russell's paradox. We can preempt this by assigning to each universe an arbitrary (but consistent) position in a sub-hierarchy, but this quickly becomes tedious. Fortunately, proof assistants such as Coq handle the bookkeeping on our behalf, ensuring that the relative positions of universes is sensible. In our informal written HoTT, we take full advantage of typical ambiguity, and leave Coq to make explicit the hierarchy of universes.

2.1.3 Functions

Besides judgments, there are a number of operations we can perform on types. We begin with **functions**, which allow us to describe the relationships between objects.

Functions are primitive structures in type theory and are represented as objects of function types. They describe how, given a domain A and codomain B, one can map objects of A to objects of B:

$$f : A \to B \,.$$

We may imagine a function as a length of thread that connects two points in

space (see Figure 2.1).

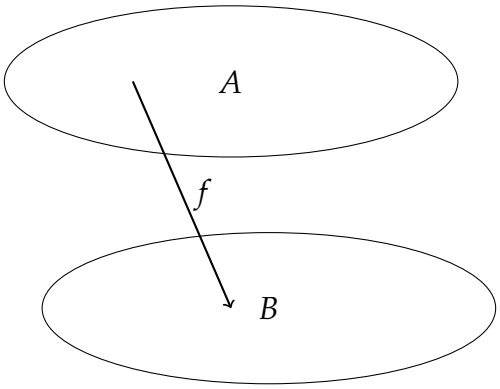

Figure 2.1: A function f with domain A and codomain B

To construct a function, it suffices to give it a name and a definition:

$$addFive : \mathbb{N} \to \mathbb{N}$$
$$addFive\ n :\equiv n + 5\ .$$

We note that the use of an infix + in the function definition is a convenience of notation. A computation is performed by **applying** a function to its arguments:

$$addFive\ 1 \equiv 1 + 5$$
$$\equiv 6\ .$$

We typically omit parentheses when we apply functions, and assume right-associativity. Thus $f\ g\ x$ could be equivalently written $f\ (g\ x)$. Parentheses can also be used to disambiguate function application: $f\ (g\ y)\ z$.

Functions of multiple inputs are represented using **currying** (named for logician Haskell Curry). A curried function $f : A \to B \to C$ is a function that takes as input an object $a : A$, outputs a function that takes an object $b : B$ and

maps it to an object $c : C$. Right-associativity holds for curried type judgments, so the same function can be written $f : A \to (B \to C)$. The application of a curried function to its arguments necessitates the use of parentheses to force left-associativity: $(f\ a)\ b$. As an example, consider the function *add* that takes two natural numbers and yields their sum:

$$add : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$

$$(add\ n)\ m :\equiv n + m\ .$$

The definition of *add* demonstrates a convenience of notation we take advantage of in this thesis: a proper definition of the function would make clear that *add* takes only one argument and outputs a function:

$$(add\ n)\ m :\equiv n \mapsto m \mapsto n + m\ .$$

But this requires the introduction of a new symbol $\mapsto$, not to mention the definitions become cumbersome as functions grow larger.

A function $f : A \to B$ is **constant** if $f\ a \equiv b$ for all $a : A$, and exactly one $b : B$. In other words, a constant function maps every input in its domain to a single output. A related term is **fiber**. Given a function $f : A \to B$, the fiber over $b : B$ is each point $a : A$ mapped to b. The alternative notation for a fiber over b is $f^{-1}\ b$, representing the inverse mapping $f^{-1} : B \to A$. A fiber is visualized as a collection of function "threads" emanating from points in A and woven into a string connected to a single point in B (as in Figure 2.2).

Functions in HoTT have a few nice properties. First, they are always total,

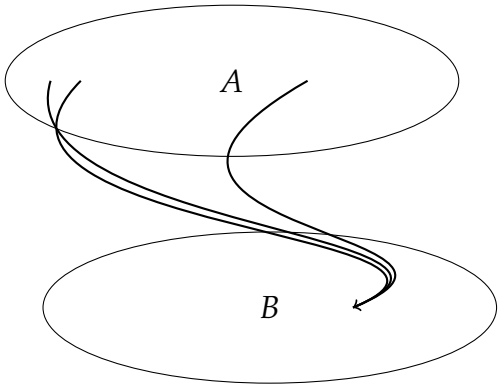

Figure 2.2: A fiber from points in A to a point in B

meaning that a function must be well-defined for each possible input. It is possible to encode partial functions in type theory, but the encoding must always be done in terms of the primitive, total kind. Functions are also always continuous (though, like partial functions, we can define discontinuous functions in terms of continuous ones), meaning—from a topological point of view—that functions preserve paths. We discuss homotopy type theory's treatment of paths in a later subsection, but rest assured that the well-behaved nature of type-theoretic functions allows us to prove theorems about spaces in remarkably elegant ways.

2.1.4 Sums and Products

Given types A and B, we may define their **sum** $A + B$. The objects of $A + B$ are either objects $a : A$ or objects $b : B$—we may imagine the sum type as containing disjoint copies of A and B, and for this reason the sum type is also referred to as the disjoint union of types. Topologically, we may imagine the object of a sum type $A + B$ as representing a space distinct from either A or B.

Indeed, this visualization clues us in to an important property of the sum of types: $A + A$ and A are not equal.

For any sum $A + B$ there are two constructors: the left injection $inl : A \to A + B$, and the right injection $inr : B \to A + B$. Accessing an object of type $A + B$ is as easy as applying an injection to an object of the relevant type: $inl\ a : A + B$ or $inr\ b : A + B$. When we need to introduce a function that makes use of a sum type, we must define it over both cases. For example, here is a function that exchanges left and right injections:

$$f : A + B \to B + A$$
$$f\ inl\ a :\equiv inr\ a$$
$$f\ inr\ b :\equiv inl\ b\ .$$

The general rule for performing computations using sum types is as follows: given a function $f : A + B \to C$, we define $f\ inl\ a$ as the application of a function $g : A \to C$ to the relevant $a : A$. Likewise, $f\ inr\ b$ is defined as the application of $h : B \to C$ to the relevant $b : B$.

The **product type** is written $A \times B$, and its objects are ordered pairs (a, b). It is similar to the classical notion of a Cartesian product and sometimes goes by the same name. Constructing an object of the product type is simple: given types A and B, we may construct an object of $A \times B$ by pairing objects from both types: $(a, b) : A \times B$.

A function $f : A \times B \to C$ is defined by a function $g : A \to B \to C$ applied to the objects $a : A$ and $b : B$ derived from the pair. This is the general computation rule for product types. To illustrate, consider defining a function

$f : A \times B \to A + B$:

$$f\ (a, b) :\equiv (g\ a)\ b\ .$$

We need a function $g : A \to B \to A + B$, which we define by case analysis:

$$(g\ a)\ b :\equiv_{inl} inl\ a$$
$$(g\ a)\ b :\equiv_{inr} inr\ b\ ,$$

where the subscripts *inl* and *inr* denote the two cases (i.e., the two ways we can define the function).

2.1.5 Finite Types

Before we introduce dependent types, we take a small detour to examine two special types, each with a finite number of objects. The first is the **empty type**, **0**. In order to construct an object of **0**, we must satisfy *zero* constructors. Contrast this with the sum type, introduced using two constructors, namely *inl* and *inr*; or the product type, constructed by specifying two objects to be grouped into a pair. The empty type can be visualized as a space with no points—an absence of shape that does not restrict movement. Because there is no restriction on where we can go in empty space, we can freely travel to any other space and observe points there. In type-theoretic terms, this corresponds to an ability to map a object of **0** to any object at all:

$$f : \mathbf{0} \to A\ .$$

We may also map a point to the empty space

$$g : A \to \mathbf{0} ,$$

but such a mapping necessitates that A is also empty.

The **unit type 1** is a type with exactly one object. We may visualize the unit type as a space consisting of a solitary point. We write the object of the unit type as $\bullet : \mathbf{1}$, and specify that any computation using the unit type need only satisfy one case: that of $\bullet$.

The empty and unit type appear frequently in type theory and can be used to construct other finite types. For example, **2**, the type with two distinct objects, can be constructed from the sum $\mathbf{1} + \mathbf{1}$, and the injections $inl\ \bullet$ and $inr\ \bullet$ renamed *true* and *false*, respectively. Now we have $true : \mathbf{2}$ and $false : \mathbf{2}$, and we can represent Boolean values!

2.1.6 Dependent Types

Type families, also known as **dependent types**, are a special case of functions and take the form

$$f : A \to \mathcal{U} .$$

A dependent type takes as input an object $a : A$ and outputs some type $B : \mathcal{U}$. The classic example of a dependent type is $\mathcal{V} : \mathbb{N} \to \mathcal{U}$, which takes as its input a natural number n and outputs the type of all vectors of dimension n. Like an ordinary function, a dependent type can be constant, if it outputs the same type for every input. Also like functions, dependent types are guaranteed to

be total and continuous.

2.1.7 Dependent Product Types

The dependent product type, which we call a $\prod$ **type** for clarity, generalizes function types. A $\prod$ type is written

$$\prod_{a:A}(B\ a)$$

for some type $A : \mathcal{U}$ and dependent type $B : A \to \mathcal{U}$. The objects of $\prod$ types are functions whose codomains depend upon the value of its domains. If B is a constant dependent type, then a $\prod$ type is an ordinary function type. To illustrate, recall the dependent type $\mathcal{V} : \mathbb{N} \to \mathcal{U}$, which takes an object $n : \mathbb{N}$ and outputs the type of all n-dimensional vectors. We introduce a function

$$f : \prod_{n:\mathbb{N}}(\mathcal{V}\ n)\ ,$$

which may be applied to an argument to yield a specific n-dimensional vector $\vec{v}$.

2.1.8 Dependent Sum Types

Dependent sum types, or $\sum$ **types**, generalize the product type. The objects of $\sum$ types are pairs (a, b), where the type of b depends on the value of $a : A$. We write $\sum$ types as follows:

$$\sum_{a:A}(B\ a)\ ,$$

for type $A : \mathcal{U}$ and dependent type $B : A \to \mathcal{U}$. If B is constant, the $\sum$ type is an ordinary product type. Our type of n-dimensional vectors can be used to create a $\sum$ type of pairs indexed by the natural numbers:

$$(n, \vec{v}) : \sum_{n:\mathbb{N}} (\mathcal{V}\ n) .$$

2.2 Proof and Logic

2.2.1 The Curry-Howard Correspondence

There is a correspondence between types as understood in type theory, and logical propositions. This relationship goes by many names—the **Curry-Howard correspondence**, propositions as types, and proofs as programs, to name a few—and states an equivalence between computer programs, proofs of propositions, and objects of types. Homotopy type theory allows us to take the equivalence a step further, and consider a proposition as a space whose points are the proofs, or **witnesses** of that proposition.

Homotopy type theory is therefore a constructive mathematics; that is, proving a theorem means introducing a type that corresponds to some proposition, then specifying an algorithm to construct an object of that type. The algorithm can be verified computationally, so proofs in HoTT may be written both informally, in standard mathematical prose, and formally, as computer programs.

2.2.2 Identity Types

Judgmental equality is well and good for *defining* the equality of two objects, but what if we want to *prove* that two objects (or types) are equal? For this we must introduce the **identity type**. Given a type A, and two objects $a, b : A$, we represent a propositional equality between them by writing $a = b$. A propositional equality can only be introduced between objects of the same type, and the collection of identities between all the objects of A are represented by the dependent type $Id : A \to A \to \mathcal{U}$. An object of the identity type is a witness of the equality; the object $p : a = b$, for example, can be considered a "proof" that a and b are equal. Topologically, we may consider p a path whose start and endpoints are a and b, respectively.

There is a special case of identity, called **reflexivity**, which describes the equality $a = a$. We give the name $refl$ to a witness of the $\prod$ type $\prod_{a:A}(a = a)$, which states that each object of a type A is equal to itself by reflexivity. The function $refl$ also comes with computational rule, known as the principle of induction for equality—or simply **path induction**, if we indulge the topological analogy—which states that in order to prove properties of $a = b$, it is sufficient to consider only the case $refl_a$, or the case where $a = a$.

2.2.3 Propositions

There are a great many ways to embed logic in homotopy type theory, though here we favor one interpretation of propositions over the others. If we allow any type to represent a proposition, conventional logic breaks down. For example, the law of double negation is incompatible with the unrestricted

propositions-as-types model.

In most cases, we want to be able to reason in the classical style, which means that our definition of proposition must restrict the amount of information a witness to a proposition is allowed to carry. Ideally, a proof of some proposition tells us nothing more than the basic fact that the proposition has a proof. To enforce this, a proposition must be **contractible** if it is inhabited; two witnesses of a proposition must be equal to each other, so an inhabited proposition is equivalent to the unit type **1**, while an uninhabited proposition is equivalent to the empty type **0**. It may be helpful to imagine the inhabited proposition type as a disc that can be shrunk—contracted—to a single point. We formulate this definition as a $\prod$ type which we call *Prop*:

$$Prop : \prod_{a:A} \prod_{b:A} (a = b) .$$

Prop takes as its input every pairing of objects from some type A and maps them to a witness of the identity type. Equivalently, we may read the type of *Prop* as "for all a and b in A, a equals b".

As an example of a simple, informal proof, we show that $x = y$ given $x \equiv y : X$.

Proof. We must construct an object of $x = y$. By reflexivity, we know that $refl_x : x = x$. By definition, $x \equiv y$, so it follows that $refl_x : x = y$. □

We often need to flatten non-propositions so that they behave as expected. We introduce the **truncation**, a type that can be formed from any type A such that the truncated type $\|A\|$ contains no information besides whether it is

inhabited or not. To construct an object $|a| : \|A\|$, it suffices to show that there is an object $a : A$.

2.2.4 Predicate Logic

We finally have the tools necessary to construct a predicate logic within homotopy type theory. Because inhabited propositions are equivalent to the unit type, we let **1** represent "true." Likewise, we let **0** represent "false," and any witness to **0** we call a **contradiction**. Note that these notions of truth and falsehood are different from the values of a Boolean type; **1** and **0** represent propositional truth and falsehood, whereas the elements of the Boolean types are mere values. This is quite different from classical mathematics, where we play fast and loose with types (and the differences between first order logic and Boolean calculus are a bit more malleable).

Logical implications are represented by functions over propositions. To prove that proposition A implies proposition B, we construct a function $f : A \to B$. To prove A is false, we construct a function $g : A \to \mathbf{0}$, which in turn means that A is empty itself, since a function that maps to the empty type is only possible if the domain is also the empty type. In particular, g is our representation of the negation of A. Conversely, a mapping from **0** to any type is always possible, representing the logical principle of "false implies anything."

Logical conjunction is represented by the product type, so $A \wedge B$ becomes $A \times B$, while logical disjunction is represented by the sum type ($A + B$ for $A \vee B$). If we wish to encode a bi-implication it is sufficient to construct the

product of two functions, one mapping from the first type to the second, and the other mapping back: $(A \to B) \times (B \to A)$.

Finally, we have the existential and universal quantifiers. If we wish to make a logical statement about all objects of a particular type, we use a $\prod$ type. For example, the type $\prod_{n:\mathbb{N}} (P\ n)$ says that for all natural numbers n, some property P holds. If instead we write $\sum_{n:\mathbb{N}} (P\ n)$, we mean that P holds for some natural number n, analogous to the phrase "there exists."

Putting the pieces together, we can now translate logical statements into types:

$$(P \Rightarrow Q) \vee (Q \Rightarrow P) :\equiv (P \to Q) + (Q \to P)$$

$$(P \wedge Q) \Rightarrow (P \vee Q) :\equiv (P \times Q) \to (P + Q)$$

$$\sim P \Rightarrow (\sim P \wedge P) :\equiv (P \to \mathbf{0}) \to ((P \to \mathbf{0}) \times P)\ .$$

If P and Q are propositions, then their product $P \times Q$ will also be a proposition. Functions over propositions also preserve their nature. But the sum and $\sum$ types add information in such a way that they are not guaranteed to be propositions as we define them. In circumstances like these, it we may need to truncate the type to reason classically.

We end the section with a demonstrative proof that given a type $\mathbb{B}$ with two elements $\top : \mathbb{B}$ and $\bot : \mathbb{B}$, then $\top \neq \bot$.

Proof. We want to prove $\top \neq \bot$. To do so, we must show how to construct an element p of the type $(\top = \bot) \to \mathbf{0}$. We define the dependent type $\mathcal{B} : \mathbb{B} \to \mathcal{U}$

by the following equations:

$$\mathcal{B}\,\top :\equiv \mathbf{1}$$

$$\mathcal{B}\,\bot :\equiv \mathbf{0}\,.$$

Recall that $\mathbf{1}$ has only one element, namely $\bullet$, and $\mathbf{0}$ has no elements at all. Because $\mathcal{B}\,\top \equiv \mathbf{1}$, we may say that $\bullet : \mathcal{B}\,\top$. Assume that $\top$ and $\bot$ are equal, meaning that there exists some arbitrary $p : \top = \bot$. Then we can substitute $\bot$ for $\top$ in $\bullet : \mathcal{B}\,\top$ to derive an element $\bullet : \mathbf{0}$, a contradiction. Our "proof" that $\top = \bot$ was an arbitrary p, so we can apply the process above to any witness that $\top = \bot$. In other words, we have an element $p : (\top = \bot) \to \mathbf{0}$. □

2.2.5 The Univalence Axiom

Vladimir Voevodsky's Univalence Axiom is perhaps the most important aspect of the homotopy type theory presented here. Rather than state it outright, we will develop several notions of equivalence and show how the property of univalence connects them.

Homotopy

Given two functions $f, g : \prod_{a:A} P\,a$ for some dependent type $P : A \to \mathcal{U}$, the **homotopy** $f \sim g$ has the type

$$\prod_{a:A} (f\,a = g\,a)\,.$$

Homotopy provides a way to equate two functions, not unlike the way in which the identity type equates elements of the same type (see Figure 2.3).

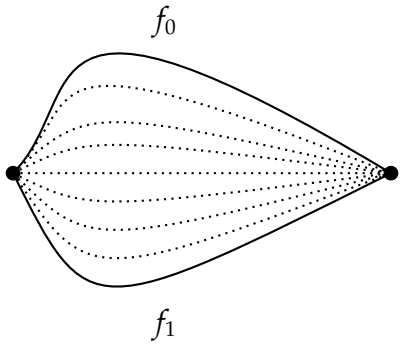

Figure 2.3: A homotopy between functions f_0 and f_1

In a traditional model, two types $A, B : \mathcal{U}$ are said to be isomorphic if there exist functions $f : A \to B$ and $g : B \to A$ such that $f \circ g \sim id_B$ and $g \circ f \sim id_A$. We may also call this relationship a homotopy equivalence, though in homotopy type theory elements of the type of homotopy equivalence are not guaranteed to be inhabited by one unique element. Structures of higher dimension may have many non-trivial homotopies, thus the formulation of homotopy equivalence over f as a triple (a function g, and homotopies $f \circ g \sim id_B$ and $g \circ f \sim id_A$) we call a **quasi-inverse**. If we want a better-behaved notion of equivalence, we must turn to functions and their fibers.

Equivalence

A type A is a **singleton** if for some $a : A$, a and b are equal for all $b : A$:

$$singleton\ A : \sum_{a:A} \prod_{b:A} (a = b) .$$

Here the function *singleton A* receives a type A and constructs a pair that contains an $a : A$ and a function that maps every $b : A$ to the identity $a = b$. In other words, a type A is a singleton if there exists some element such that for every other element b, $a = b$.

Previously, we introduced the fiber of a function $f : A \to B$ as the type of points $a : A$ mapped to some point $b : B$. We can generalize this definition to include points identified with b, thus the fiber of f is written

$$f^{-1}\ b : \sum_{a:A} ((f\ a) = b)\ .$$

An **equivalence** is a function whose fibers are singletons:

$$equivalence\ f : \prod_{a:A} (singleton\ (f^{-1}\ a))\ .$$

We let *equivalence f* represent the type of function that receives a function f, and maps every $a : A$ to the singleton fibers of f. We say a function $f : A \to B$ is an equivalence if, for every $b : B$, the fiber $f^{-1}\ b$ has one element only. We can collect the ways in which two types A and B are equivalent in the type $A \simeq B$, much like how $A = B$ can be said to be the collection of ways in which the types are identified.

Univalence

Consider the following function:

$$idToEq : (A = B) \to (A \simeq B)\ .$$

Given a proof that A equals B, *idToEq* produces a proof that A and B are equivalent. The **Univalence Axiom** tells us something rather special about this function: it is an equivalence.

$$(A = B) \simeq (A \simeq B)$$

In plainer English, the Univalence Axiom tells us that the identity on the left can always be mapped to the equivalence on the right such that the mapping is also an equivalence. This is remarkable for two reasons; the first is that it gives us a formal way to say that equal things are equivalent; for example, the bijection between the natural numbers and the integers is not just handy function, it is an equivalence.

The second reason is that it implies the existence of multiple possible proofs of identity between structures, beyond reflexivity. Because it is often useful to work with a more traditional mathematics within HoTT, we call those types whose elements are identified by reflexivity alone **sets**, defined by the following type:

$$set\ A : \prod_{a,b:A} (Prop\ a = b)\ ,$$

given the type *Prop* from Section 2.2.3. Under the context of univalence, we no longer need to specify when structures are, for example, equivalent "up to homotopy," since homotopy and identity are equivalent.

Chapter 3

Computation

The project of studying computation requires first and foremost a model to study. A **model of computation** is a mathematical object that defines what it means to be a computer. It is meant to capture the notion of computation in a purely mathematical sense, so it is useful for determining the limits of a computer given an infinite amount of time and memory. Depending on what model is used, the features of a computer can vary significantly. Fortunately, the various classes of computational models form a natural hierarchy and can be presented in a logical order [13].

This chapter gives one such presentation. We discuss formal languages, automata theory, and the notion of infinity with respect to computation. The final section presents evidence for the usefulness of homotopy type theory (and type theory generally) in formalizing different models of computation.

3.1 Formal Languages and Automata

A **formal language** is a set of symbols (the **alphabet**) paired with a set of strings over that alphabet. For example, given an alphabet with two symbols {`0, 1`}, we can define a set of strings such that every string has an even number of zeroes. The resulting language is one in which every string is a binary string with an even number of zeroes. `0011010`, `00001011110`, and `111` are all examples of strings in this language, as is ϵ, or the **empty string**. The empty string contains no symbols, and it is a string over every alphabet.

The number of symbols can be as small or as large as we need it to be, so long as it remains finite. An alphabet with thousands of symbols may not be of much practical use to a human being, but it could be perfectly feasible for a computer. However, we tend to restrict our use of symbols to a few, or even two. The use of binary reflects the manner in which most physical computers are built and makes defining functions over the alphabet simpler.

Formal languages fall into various groups depending on their behavior, and each linguistic class is associated with a corresponding class of abstract machines called **automata**. An automaton consists of some number of **states** and a function that determines how to move from state to state on a given input. The automaton receives a program in the form of a string, changes its state, and possibly determines whether a given input is valid or not, depending on what state the machine is in when the input has been consumed.

3.1.1 Regular Languages and Finite Automata

The simplest type of automaton is called a **finite automaton**, and it consists of exactly the following:

- a set of states Q
- an alphabet Σ
- a transition function $\delta : Q \times \Sigma \to Q$
- a starting state $q_0 \in Q$
- a set of accept states $F \subseteq Q$

A computation involving a finite automaton begins at q_0 and transitions states upon reading each symbol of the input string. We say the automaton **accepts** a string s if, after consuming every symbol in s, the machine is in one of the accept states. Likewise, we say a formal language is **recognized** by a finite automaton if every legal string in the language is accepted and every illegal string is rejected [13].

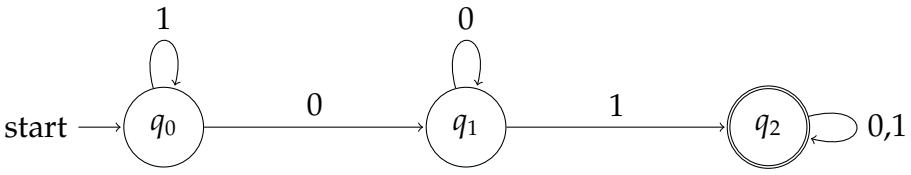

Figure 3.1: An example finite automaton

Figure 3.1 gives a diagram for a finite automaton that accepts the language of binary strings with at least one 0 and one 1 in that order. Arrows indicate the transitions between states; the labels next to each arrow indicate the symbol

that triggers the transition. The state q_2 is the accept state, shown here with a double circle, and the automaton accepts only when it is in the accept state and the entire input string has been consumed.

Finite automata are ideal machines for computing with very little memory, since they only need to keep track of their current state and symbol. Vending machines are a good example of this: The alphabet of a vending machine consists of various coins, and we accept any combination of coins that brings us to a particular sum.

A **regular language** is any language that is recognized by a finite automaton. The following are all examples of regular languages:

- the set of strings over $\{0, 1\}$ containing an even number of zeroes
- the set of strings over the English alphabet containing the substring "HoTT"
- the **empty language** $\varnothing$

The empty language is the language that contains no strings, and it is accepted by the automaton with no accept states. Note the difference between $\varnothing$, which denotes the empty language, and ϵ, which is a string only.

Lovely as they are, regular languages are only a tiny part of the formal language picture. Even simple, very useful languages like the language containing strings of balanced pairs of parentheses are not regular. No amount of twisting and pulling a finite automaton is going to coerce it into recognizing strings such as `()(()(()))` while rejecting strings like `(()))`. For such a

language we require additional resources and a more powerful rule for building strings out of symbols [13].

3.1.2 Context-Free Languages and Pushdown Automata

Balancing parentheses requires an ability to count, which finite automata sorely lack. But if we take a finite automaton and add a simple memory structure—a **stack**—we create a new type of computer.

The **pushdown automaton** has states, a start state, some accept states, and a transition function. But every transition from one state to another conveys information about what to write or remove from the stack. The automaton that recognizes our balanced parentheses language simply pushes a symbol to the stack for every open parenthesis it reads, and pops a symbol for every closed parenthesis. The machine is only allowed to accept when the entire input has been consumed, the stack is empty, and the automaton is in an accept state.

The precise definition of a pushdown automaton is quite similar to that of a finite automaton:

- a set of states Q
- an input alphabet Σ
- a stack alphabet Γ
- a transition function $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\}) \to \mathcal{P}(Q \times \Gamma)$, where $\mathcal{P}$ represents the power set and $\{\epsilon\}$ is the language containing only the empty string

- a starting state $q_0 \in Q$
- a set of accept states $F \subseteq Q$

Although we call the stack alphabet Γ, there is no reason why the stack and the input cannot use the same alphabet. The difference in notation is purely one of convenience, since a stack may not require the use of more than one symbol [13].

Note also the use of the power set $\mathcal{P}(Q \times \Gamma)$ in the type of the transition function. This is because our definition of a pushdown automaton allows **non-deterministic** behavior, i.e., for any state there can be several possible transitions. When a non-deterministic automaton has multiple possible transitions at any given time, it will take all of them simultaneously. Imagine a computer program that uses multi-threading to perform multiple operations simultaneously. So long as one "thread" ends in the accept state, the entire machine will accept its input. Non-deterministic pushdown automata are strictly more powerful than their deterministic cousins, who accept only a proper subset of the context-free languages. This is in contrast to finite automata, where introducing non-determinism does not increase computational power.

The class of **context-free languages** is exactly the class of languages recognized by pushdown automata. Context-free languages are common objects in the study of programming languages because the syntax of many languages are readily represented using **context-free grammars**, which encode context-free languages. The following are examples of context-free languages:

- all regular languages

- the language over $\{0, 1\}$ containing n zeroes followed by n ones.
- the language of balanced parentheses

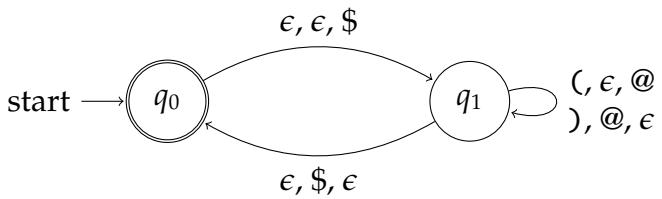

Figure 3.2: A pushdown automaton for balanced parentheses

Figure 3.2 gives an example automaton that accepts strings of balanced parentheses. Each transition has three symbols: the first is the input string symbol that triggers the transition. The second symbol is part of the stack alphabet, and is popped from the top of the stack. The third symbol is then pushed to the stack. Note that a transition only occurs if the input symbol is matched, and the stack operations are possible.

In our example, the stack uses \$ to mark the beginning of the stack, and @ to indicate that an open parenthesis has been read. The machine will only transition back to the accept state once all the @s have been removed, there is no more input, and it is possible to pop the \$ from the stack. Lastly, the ϵs in the first position indicate that the state is simultaneously in both the pre-transition state and the post-transition state; in the second and third positions they indicate popping and pushing the empty string, respectively.

The class of context-free languages contains the class of regular languages, since the pushdown automaton could just ignore its stack. Still, context-free languages are not sufficient to capture every possible program. For example,

the language over $\{0, 1, 2\}$ containing n zeroes followed by n ones followed by n twos is not context-free [13]. In order to recognize this language, we must construct yet another automaton—a **Turing machine**.

3.1.3 Turing Machines and Recursively Enumerable Languages

The language described above belongs to a class of languages called **decidable** or, equivalently, **recursive**. (In fact, it also belongs to the class of context-sensitive languages, which lies between context-free and decidable. But languages that are decidable but not context-sensitive are challenging to describe). Decidable languages are languages that are **decided** by a Turing machine. That is, for every string in a decidable language, a Turing machine will either terminate in an accept or reject state.

Turing machines are the most complex machines we have seen thus far. They are state machines, like the other automata, but with the addition of memory in the form of an infinite tape. The tape is divided into cells, which the Turing machine can read and change using a **read-write head** much like the needle of a record player.

The formal definition of a Turing machine is as follows:

- a set of states Q
- an input alphabet Σ
- a tape alphabet Γ
- a symbol $\gamma \in \Gamma - \Sigma$ that represents an empty cell

- a transition function $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$, where L and R represent shifting the read-write head of the machine left or right, respectively
- a starting state $q_0 \in Q$
- an accept state $q_{accept} \in Q$
- a reject state $q_{reject} \in Q$

Note that [13], unlike the pushdown automaton, the tape alphabet Γ must be distinct from the input alphabet Σ, because we require a special blank cell symbol γ, distinct from any symbol in Σ.

Computation on a Turing machine involves writing some input to the infinite tape and placing the read-write head on the first symbol of the input. The machine reads the symbol on the cell it is currently looking at, then changes state, moving the read-write head and altering the contents of a memory cell. The computation terminates when the machine enters the accept or reject state. We can even interpret the contents of the tape as the machine's "output," if the computation we need to perform yields a result.

Perhaps most importantly, if a Turing machine never enters a halting state, it never terminates. Figure 3.3 gives an example Turing machine that, given a tape of zeroes, will write as many ones as possible to the tape before halting. Like a pushdown automaton, each transition has three symbols. Here, the first symbol represents the symbol read from the input tape, and the second the symbol that replaces that input on the tape. The third symbol tells the Turing machine whether to move the read-write head to the right or the left before transitioning to the next state. Note that this particular machine has no reject

state. This is because we care only that the machine halts, not necessarily rejects or accepts.

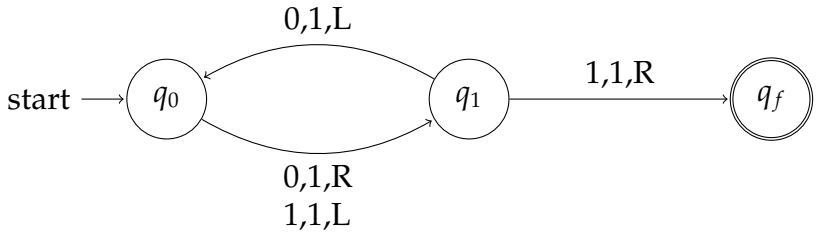

Figure 3.3: A three-state Turing machine

Unlike the other automata we have seen, a Turing machine is content to keep churning, crunching numbers and writing to its tape for all time. This raises the question: what does it mean for a Turing machine to *recognize* a language?

A language is **Turing-recognizable** if, for all inputs in the language, the Turing machine accepts. Strings not in the language are either rejected or cause the Turing machine to loop forever. Readers may notice that decidable languages do not permit the looping behavior of a fully-fledged Turing machine. There are, however, some Turing-recognizable languages that are not decidable, and these—along with all the other language classes we have seen thus far—we call the **recursively-enumerable** languages.

Recursively-enumerable languages are as powerful as formal languages get, at least in the context of computation. We cannot exceed the power of a Turing machine because the Turing machine *defines* what it means to be computable. Naturally, we might wonder exactly how many possible programs there are. An infinite number, of course, but *how* infinite, exactly? Are there formal languages that exceed the power of the recursively

enumerable? As it happens, there are.

3.2 The Sizes of Infinity

Given two infinite sets, how can we tell when one set is larger than the other? For that matter, how can we tell that they are the same size? Is it even possible for two infinite sets to have different sizes at all? As it happens, there are a number of methods to satisfactorily answer all these questions, the most intuitive being to use bijections, that is, provide a method that pairs the elements of one set with the elements of another such that any element from one set always has a companion from the other. An example of two distinct, yet equal-size, sets are the natural numbers and the integers:

$\mathbb{N}$	$\mathbb{Z}$
0	0
1	1
2	-1
3	2
4	-2
$\vdots$	$\vdots$

We can always pair the next natural number with the next integer in sequence, without leaving an element out from either set. The same result can be inductively proven, though it usually suffices to simply construct a pairing, as above.

The natural numbers define the smallest possible infinity, $\aleph_0$, also called

countable infinity. Any infinite set can be shown to be countable if its elements can be placed in a one-to-one correspondence with the natural numbers.

As it happens, the class of recursively enumerable languages is countably infinite. We can show this intuitively, by enumerating every possible computer program in binary:

$\mathbb{N}$	Turing machine programs
0	0
1	1
2	10
3	11
4	100
⋮	⋮

Put another way, every possible computer program written in any programming language must compile to binary machine language. This binary code can be thought of as a single, very large natural number. Thus the enumeration of the natural numbers is tantamount to the enumeration of all programs.

Showing that there are infinities larger than the natural numbers is an interesting challenge. Perhaps the easiest method to show that, for example, the real numbers, are **uncountable** is to use Cantor's diagonalization argument.

Consider the following hypothetical correspondence between the natural numbers and the real numbers:

$\mathbb{N}$	$\mathbb{R}$
0	0.012345...
1	0.123457...
2	4.012245...
3	2.353537...
4	0.333333...
⋮	⋮

We could keep finding pairs like this forever, but do we actually have a one-to-one correspondence? As it happens, writing out every single real number like this is more than a Herculean task, it is impossible.

Given an infinite table of correspondences, we can always create a new decimal number that is not represented. We can construct an unrepresented number between 0 and 1 by selecting the first digit of the first real in our table, the second digit of the second real, and so on. Then we change each digit to either a 5 (if it was not already a 5) or a 4 (if it was previously a 5), so that we have a new number in every place after the decimal point. The selection of 5 and 4 are arbitrary, but it allows us to avoid the digits 0 and 9 entirely and thus prevent concerns such as 2.000... from arising. The method is illustrated below:

$\mathbb{N}$	$\mathbb{R}$
0	$0.\underline{0}12345...$
1	$0.1\underline{2}3457...$
2	$4.01\underline{2}245...$
3	$2.353\underline{5}37...$
4	$0.33333\underline{3}...$
$\vdots$	$\vdots$

Our new number begins $0.55545...$, and we are guaranteed that it is different from every number represented in our infinite table by at least one digit. Thus the real numbers cannot be placed into a correspondence with the natural numbers, and they belong to a size of infinity all their own.

We can use a similar line of reasoning to show that some languages are not Turing-recognizable. To do so, we first show the following:

Proposition 1. *The power set of any countably infinite set is uncountable.*

Proof. First, assume for the sake of contradiction that $\mathcal{P}(\mathbb{N})$ is countable. We can list the subsets of the natural numbers as $N_0, N_1, N_2, ...$ so that there is a subset N_i for all $i \in \mathbb{N}$. We construct the set M that contains all the numbers i that are not members of their respective N_i subset. M is therefore a subset of $\mathbb{N}$, and $M = N_j$ for some $j \in \mathbb{N}$. But this leads to a contradiction, since if j is in M then it is not in M, and vice-versa. Hence $\mathcal{P}(\mathbb{N})$ is not countably infinite. □

We know that the set of all languages is the power set of the set of all strings over a finite alphabet, Σ^*. Because Σ^* is a countable set, it follows that the set of all languages is uncountably infinite. Because we can place the set of

all Turing-recognizable languages in correspondence with the natural numbers, there must be languages in $\mathcal{P}(\Sigma^*)$ that are not Turing-recognizable [13]. In fact, there must be an uncountably infinite number of them!

3.3 Computation in HoTT

Determining whether or not homotopy type theory supports a computational interpretation is still an open area of research. The type theory as presented in Chapter 2 does not admit a computational interpretation; performing a computation over an element of univalence is undefinable under the current formalisms, though work in other theories such as cubical type theory looks to solve this problem [1].

Reasoning about computation in HoTT is another story entirely. It is possible to postulate and reason about models of computation internally, including those problems concerning "impossible" structures, such as the function encoding the solution to the halting problem, the problem of determining whether a computer program will ever terminate on a given input (such a function must be empty, like types such as $1 = 2$).

3.3.1 Turing Machines as Functions

Turing machines provide, as the *de facto* definition of computation, a good starting place. Appendix A provides a Coq implementation of a Turing machine, adapted from [4], and a corresponding 3-state **busy beaver game** implementation. A busy beaver game comprises a binary-alphabet Turing

machine that attempts to write the maximum number of ones to its tape using as few states as possible. Additionally, the initial tape must consist only of zeroes and the machine must eventually halt. The machine in Figure 3.3 is a 2-state busy beaver, since we do not count the halting state when constructing busy beavers.

The Turing machine implementation in Appendix A is composed of several small components:

- an inductive type representing the three possible movements of a read-write head
- a transition function
- a set of states
- a type representing the Turing machine's alphabet.

Lastly, there is a function that glues the various components together. This function (given in the appendix as `TM`) receives a tape of symbols, as well as the definitions of each component, simulates each step of the machine (each application of the transition function) and returns a tuple containing an updated tape and machine state.

The busy beaver function follows this structure nicely. We give an inductive type containing four distinct elements as states (three "computing" states and one halting state) and use a Boolean type as our alphabet. The transition function pattern matches according to the 3-state busy beaver algorithm, and the machine as a whole is set to halt only when the machine is in the halting position. The program is certainly no **Universal Turing machine**

(a Turing machine that can simulate any Turing machine on arbitrary input) [13], but it should convince the reader of the possibility to reason type-theoretically about computation.

Of course, if we can reason about computation, we ought to be able to discuss its properties. For example, let $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ be defined by applying some program encoded as natural number to an argument (also encoded as a natural number). The result of the computation is the element of $\mathbb{N}$ that f maps its inputs to.

This representation holds for program/input pairs that halt, but we need to encode the potential for partiality—meaning, in this context, the potential for a non-halting program—in our function. The simplest way to do so is to introduce a sum type $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N} + \mathbf{1}$ and send all non-halting programs to $\mathbf{1}$. Actually implementing this function is difficult, however; knowing which programs to send to the empty type is as difficult as solving the halting problem, which is to say that it cannot be solved at all.

We modify f again, this time so that it receives a third argument which represents the number of steps a given program/input pair will run for: $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \to \mathbb{N} + \mathbf{1}$. The function maps some program and its argument to a natural number if that program produces a result in some $n : \mathbb{N}$ steps, and to $\mathbf{1}$ if there is no result [15].

3.3.2 Another Approach: Turing Categories

Turing categories, originally formalized by Vinogradova, Felty, and Scott, provide yet another type-theoretic model of computation [15]. A **category** is a

structure comprising some objects connected by functions called arrows, all of which obey the axioms of associativity and the existence of an identity arrow. Instead of representing partial maps using a sum type, a category is defined that captures the properties of partial maps—eliminating the problem of "too many functions" from $\mathbb{N}$ to $\mathbb{N}$. Rather than work with the uncountably infinite maps over the natural numbers, we restrict ourselves to only that countable subset that represents the computable functions.

To this structure we add an object and a family of maps over that object representing a universal Turing machine and the application of the machine to various programs and inputs.

3.3.3 Last Thoughts

The strategies we present for modeling computation in type theory are intended as evidence for the plausibility of reasoning about computers in homotopy type theory. However, it is important to note that we still rely on interpretation to make sense of the models.

For example, the output of a Turing machine is typically represented by the contents of its tape when computation halts. But there is nothing intrinsic to the Turing machine's tape that makes its suitable for output—we could just as easily add an "output terminal" to our definition of Turing machine to capture the relevant notion.

Likewise, in the work of Chapter 5, we make some assumptions about the workings of our imaginary machines. For example, we assume that a program is simply a function that takes a value from a type A to another value of type

A. We also assume the existence of an external "user" to influence the behavior of our program in cases where interactive input is necessary.

By giving several interpretations of our work as it applies to common problems in reasoning about effects, we hope to develop a body of evidence for the usefulness of the theory. Future work may be necessary to rigorously prove the efficacy of the theories, or to narrow its scope.

Chapter 4

The Coq Proof Assistant

Coq is a formal proof assistant and programming language created in 1984 and based on Thierry Coquand's calculus of constructions [14]. It provides a library of tools that can be used to validate software, formalize mathematical proofs, and occasionally encode entire branches of mathematics.

The purpose of this chapter is to familiarize readers with the style and meaning of Coq proofs. While Coq supports a bevvy of related software for validating programs, writing proofs, and compiling results into libraries for future use, we aim only to cover the basics of reading the proofs themselves.

4.1 Inductive Types

Inductive types are one of Coq's most primitive constructions. An inductive type always begins with the keyword `Inductive`, followed by an identifier and definition. As an example, take the inductive definition of the natural numbers:

```
1  Inductive nat : Set :=
2    | O : nat
3    | S : nat -> nat.
```

We specify that `nat` belongs to the `Set` type, which in Coq is the smallest possible universe of types. The definition consists of two constructors: the base case of the natural numbers, zero (written in Coq as the capital letter `O`), and the function that takes a natural number to its successor `S`.

Under this definition, we expect that any element of `nat` will either be zero or the application of a function from `nat` to `nat`. Thus the number one is `S O`, the number three is `S (S (S O))`, and the number six is `S (S (S (S (S (S O)))))`. The Coq standard library allows us to write natural numbers using decimal digits, but it is important to keep in mind that any large number is treated by Coq as a sequence of successor functions.

Coq also does most of the legwork in determining how any given inductive type ought to work. When a new type is specified, the proof assistant automatically adds several functions to the current environment. These represent the expected properties of the natural numbers. For example, in order to prove any p about `nat`, it suffices to prove p holds for `O` and `S`. Coq will also treat `O` and `S` as different under any circumstance, encoding the property that the constructors of any inductive type are never equivalent [14].

4.2 Basic Proofs

As a simple example of how proving statements in Coq works, let us prove that 4 and 4 are equal.

```
1  Theorem FourIsFour : 4 = 4.
2  Proof.
3    reflexivity.
4  Qed.
```

As before, our program begins with a title-case command. In this case, we use `Theorem` to identify the name `FourIsFour` with a proposition `4 = 4`. The `Proof` command tells Coq that the lines that follow construct an element of type 4 = 4. The only line in the proof itself is `reflexivity`, which simply asserts that 4 equals 4 under the simplest principle of equality. Finally, we write `Qed` to end the proof and bind the construction to the identifier.

As a slightly more complex example, we prove that A and B implies A.

```
1  Theorem AxBImpliesA : forall A B : Type, (A * B) -> A.
2  Proof.
3    intros A
4           B
5           p.
6    exact (fst p).
7  Qed.
```

As in an informal type-theoretic proof, we expect our construction to be a function of type $\prod_{A,B:\mathcal{U}} (A \times B) \to A$. The first line of the proof says exactly this,

although we substitute `forall` for $\prod$, and `Type` for $\mathcal{U}$.

The `intros` keyword introduces named variables representing the information we have at our disposal. Here, we introduce three variables `A`, `B`, and `p`, representing the types A, B, and $A \times B$, respectively. The tactic is akin to saying "assume we have types A and B, and an element $p : A \times B$" in an informal proof.

Lastly, we project an element of A out of the product by applying the `fst` (for "first") function to our element `p`. We mark the result with `exact` to tell Coq that the projection gives a proof of the goal—in this case, an element of A.

A full English translation of this proof might be: assume types A and B, and an element $p : A \times B$. Such a p is of the form (a, b) for elements $a : A$ and $b : B$, so we have an element $a : A$. Therefore, $A \times B$ implies A.

4.3 Records

Coq provides several built-in macros to simplify the process of defining more complex types. For example, the $\sum$ type can be defined using induction alone, but doing so for even simple dependent pairs can be tedious at best.

Record types allow us to represent dependent sums as lists of elements. We might, for instance, desire to prove theorems about the rational numbers, which necessitates a constructive representation of the rationals in Coq. Such a representation would likely look something like:

```
1  Record rat : Set := {
2    p : nat
```

```
3       ; q : nat
4       ; not_zero : (gt q 0)
5     }.
```

Here `p` and `q` are natural numbers and `not_zero` is a proof that `q` is greater than zero. Note that we do not explicitly define the negative rationals by this definition because the negative rationals can be placed in a correspondence with the non-negative rationals, i.e., the properties of one can be proven to apply to the other. Rationals that reduce to one another can also be proven to be equivalent, even if it is not made explicit by the definition. To define an element of this type in Coq, we first select our `p` and `q`, 1 and 3 in this example, which will stand in for the numerator and denominator of our rational number. Then, we prove that 3 is greater than 0:

```
1   Theorem ThreeGreaterThanZero : (gt (S (S (S O))) O).
```

Our theorem can be proved on a single line since it requires only the expansion of the definition of three and the application of the `gt`, or "greater than" function.

With all the pieces in hand, we can define $\frac{1}{3}$:

```
1   Definition third := (Build_rat 1 3 ThreeGreaterThanZero).
```

The `Definition` command begins the definition of some object. Unless a new constructor is explicitly defined, Coq automatically generates a record constructor with the name `Build_<IDEN>`, where `<IDEN>` is the name of the user-defined record [14].

4.4 Functions

Function definitions are straightforward, though they are remarkably diverse in their appearances in Coq proofs. We present the most straightforward version here, though the reader is encouraged to read the documentation for more information on the various styles of function definition.

The following example defines the Boolean operation `and`:

```
1  Definition and (p q : bool) :=
2    match p, q with
3      | true, true   => true
4      | true, false  => false
5      | false, true  => false
6      | false, false => false
7    end.
```

Function parameters are placed in parentheses after the identifier. In this case, we are defining the logical operator `and`, which receives two arguments of type `bool`, called `p` and `q`.

The `match` keyword begins a pattern-match block over the specified variables. Since we have two variables with two possible values each, we must define the function output on four possible inputs. If we leave a case out, Coq will not accept the function definition. The `match` block is ended with the `end` keyword.

4.5 HoTT in Coq

As a final note, we point the reader in the direction of the HoTT library for Coq. It re-implements much of the standard library, and adds it Univalence as well as support for HoTT-specific structures. For the remainder of this paper, we assume the library is installed and in use—the code given in the Appendix accompanying this thesis will not run without it. Source code and installation instructions can be found at [8].

Chapter 5

Actions and Effects

Our definition of a type of effectful actions is motivated by an attempt to determine some small (ideally minimal) class of data points that give way to a useful computational interpretation. We know that given a type of effects Φ, a program of type $A \to \Phi \to A$ satisfies the general rules of **referential transparency**, which states that any program input can be replaced with an equivalent value without altering the output, so long as any $\phi : \Phi$ encompasses fully the computational effect we want to capture. This requires some discipline on the part of the programmer, though most of the legwork is in defining the effects to be captured by any one program. Much like defining a new datatype, once an action is defined, it may be used freely and without much fuss.

What makes an action an action? In order for a computation to take place, we need a set of values A to compute. We use sets here in order to simplify proofs of identity; we know that in order to show equality of elements in a set, it is sufficient to show identity by reflexivity.

We also require a type of effects parameterized by A. The elements of this type represent the possible effects any action might induce. Because actions can transform data as well as alter state, we need some guarantee that the structure of the type A is retained. This way, an effect can be inserted into an existing program without altering the outermost computation—the functional equivalent of inserting a print statement mid-calculation.

Finally, an action ought to have some way of transforming itself into another action. If we consider the state of a program to be the current state of a board game—chess, for example—then this transformation operation represents the next legal move or moves. We stipulate that this operation alone is allowed to modify the $a : A$ bound to any effect $\phi : \Phi$. That is, the binding and immediate evaluation of a to any ϕ should not meaningfully alter a in any capacity. Because we aim to treat effects and actions as data in and of themselves, we cannot allow an effect to modify the data parameterizing it except by explicit transformation of the effect at hand.

With these restrictions in mind, we give the following definition of an effectful action:

Definition 1. *Given a set A of values and a type $\Phi(A)$ of potential effects parameterized by A, an **action** over $\Phi(A)$ comprises:*

- *A function bind* $: A \to \Phi(A)$.
- *A function eval* $: \Phi(A) \to A$.
- *A function transform* $: \Phi(A) \to \Phi(A)$.
- *For all* $a : A$, *we have eval* $(bind\ a) = a$.

The most immediate result of this definition is that it is possible to map any function over A to a function over $\Phi(A)$:

Proposition 2. *For all functions $f : A \to A$, there is a corresponding function $f_{\Phi(A)} : \Phi(A) \to \Phi(A)$.*

Proof. We define $c : \prod_{f:A\to A} \Phi(A) \to \Phi(A)$ by the following equation:

$$c\ f :\equiv \lambda\phi.bind\ (f\ (eval\ \phi))\ ,$$

for $\phi : \Phi(A)$. □

We use the λ-abstraction to mark the creation of an anonymous function; the notation

$$\lambda x.y$$

indicates a function that takes some x and returns some y. In the proof above, we define our function as one that takes an existing function f and applies it to the result of an *eval*, then binds the result to an effect.

5.1 Applications

We turn now to the interpretive work required of an analysis of effects. The goal of these examples is to show the basic utility of a type of effects, and to convince the reader that the class of actions encompasses at least those computational effects that are the most "fundamental," e.g., writing to memory, I/O, and exception handling.

5.1.1 Identity Action

The simplest action possible is no action at all—the only "effect" such an action has is to return its bound value. The Identity Action's type of effects $\mathcal{I}(\mathcal{A})$ is defined inductively as a type with one constructor: $\mathit{Return}\ a : \mathcal{I}(A)$. We give the following function definitions:

- $\mathit{bind}\ a :\equiv \mathit{Return}\ a$.
- $\mathit{eval}\ (\mathit{Return}\ a) :\equiv a$.
- $\mathit{transform}\ (\mathit{Return}\ a) :\equiv \mathit{Return}\ a$.

We also require the following proof:

Proposition 3. *For all $a : A$, $\mathit{eval}\ (\mathit{bind}\ a) = a$*

Proof. In order to show that $\mathit{eval}\ (\mathit{bind}\ a) = a$ for all $a : A$, we must construct a function of type

$$\prod_{a:A} \mathit{eval}\ (\mathit{bind}\ a) = a\ .$$

By definition we see that $\mathit{eval}\ (\mathit{bind}\ a) \equiv \mathit{eval}\ (\mathit{Return}\ a) \equiv a$, and thus $a = a$. Hence, for any $a : A$ we have $a = a$. □

The natural interpretation of a program $p : A \to \mathcal{I}(A)$ is one that simply wraps a term $a : A$ for later use. The existence of an identity is potentially useful as a base case for recursive operations, or as an identity with respect to some composition of actions.

5.1.2 Interactive Input

Computer programs often require the user to input some value before evaluating the final result. We might represent such a program with a function $p : In(A) \to A$. We define $In(A)$ inductively as a type with two constructors:

$$Init\ a : In(A)$$

$$Input\ a : In(A)\ .$$

Let $Init\ a$ represent an initial state, where no input has yet been performed. We give the following function definitions:

- $bind\ a :\equiv Init\ a$.
- $eval\ (Init\ a) :\equiv a$,
  $eval\ (Input\ a) :\equiv a$.
- $transform\ (Init\ a) :\equiv Input\ a^*$,
  $transform\ (Input\ a) :\equiv Input\ a$

The requisite proof of invariance under *eval-bind* composition is identical to the last. The $Input\ a^*$ is marked with an asterisk to signal that the evaluation of $transform$ may yield a different $a : A$, depending on what the user entered. For the sake of the equational theory, we only care that a^* is an element of A, not what its actual value is—the condition that a is invariant under $eval - bind$ composition is only guaranteed to be true if no other functions are applied to an effect before it is evaluated.

5.1.3 Exception Handling

The type of exceptions is also an inductive type, called $\mathcal{E}(A)$ with two constructors:

$$\mathit{Except}\ a : \mathcal{E}(A)$$

$$\mathit{Return}\ a : \mathcal{E}(A)\ .$$

We let *Return a* represent an unmodified value, where no exception is thrown. Conversely, *Except a* stands in for a problematic value of A and triggers an exception when it is evaluated.

The first two function definitions look familiar:

- $\mathit{bind}\ a :\equiv \mathit{Return}\ a$.

- $\mathit{eval}\ (\mathit{Return}\ a) :\equiv a$,
  $\mathit{eval}\ (\mathit{Except}\ a) :\equiv a$.

However, the definition of $\mathit{transform}$ depends on the type of exception we're trying to catch. The most basic implementation would simply be the identity function, and re-raise an existing exception while allowing non-exception values to pass.

More complex examples require more computational legwork. For example, if we wished to write a safe-division program with an exception handler to catch zeroes, we could first pass the arguments of our division as a pair to *bind*, then $\mathit{transform}$ them such that any pair whose second element is

zero becomes an instance of *Except*:

$$transform\ Except\ a :\equiv Except\ a$$

$$Return\ (n, 0) :\equiv Except\ (n, 0)$$

$$Return\ (n, m) :\equiv Return\ (n, m)$$

where n is any number and m is any non-zero number.

Chapter 6

Conclusion

We have briefly explored a type-theoretic method for describing computational effects in functional programs. The interpretations of Chapter 5 have, we hope, provided reasonable evidence for the viability of this model in capturing the basic semantics of effectful actions.

There still remains a good deal of work in proving the completeness of the model. Implementing this model as a design pattern in an existing language, or designing a new language based on the action type may also prove to be a useful indication that the action type has use in a real programming environment.

We have provided several examples of potential interpretations of action and effects, but additional properties of the types themselves are yet to be explored. We suspect that, ultimately, our descriptions can be identified with other models of computational effects and therefore can be formalized in terms of them, either by discovering isomorphisms or by encoding one in the other.

Appendix A

3-State Busy Beaver in Coq

```
1  Require Import HoTT.
2
3  (* TM Definition adapted from casperbp's coq file, and lecture notes
4     by Andrea Asperti & Wilmer Ricciotti *)
5
6  CoInductive CoList (A: Type) := CONS (a:A) (t:CoList A) | NIL. (* List type for tape *)
7
8  Arguments CONS [A] _ _.
9  Arguments NIL [A].
10
11 CoInductive Delay A := HERE (a:A) | LATER (_:Delay A).
12
13 Arguments HERE [A] _.
14 Arguments LATER [A] _.
15
16 Section TuringMachine.
17
18   Variable States Symbols : Type.
19
20   Inductive Move := L | R | C. (* left right center *)
21
22   Variable delta : States * Symbols -> States * Symbols * Move. (* transition func *)
```

```
23
24  Variable Final : States -> Bool. (* Boolean function to check final state *)
25
26  Definition TM (* Definition of a TM itself *)
27              (left: CoList Symbols)
28              (right: CoList Symbols)
29              (q: States) : option (States * CoList Symbols * CoList Symbols) :=
30    match right with
31    | NIL => None
32    | CONS s t =>
33      match delta (q, s) with
34      | (q', s', move) =>
35        match move with
36        | C =>
37          Some (q', left, CONS s' t)
38        | L =>
39          match left with
40          | NIL => None
41          | CONS s'' t' =>
42            Some (q', t', CONS s'' (CONS s' t))
43          end
44        | R =>
45          Some (q', CONS s' left, t)
46        end
47      end
48    end.
49
50
51  CoFixpoint compute
52              (left: CoList Symbols)
53              (right: CoList Symbols)
54              (q: States) : Delay States :=
55    if Final q
56    then HERE q
57    else
58      match TM left right q with
59      | Some (q', left', right') =>
```

```
60          LATER (compute left' right' q')
61        | None =>
62          HERE q
63        end.
64
65  End TuringMachine.
66
67  Section BusyBeaver. (* Three-state busy beaver *)
68    Inductive BStates : Type :=
69    | a : BStates
70    | b : BStates
71    | c : BStates
72    | halt : BStates.
73
74    Definition States := BStates.
75    Definition Symbols := Bool.
76
77    (* Transition function to encode a state table *)
78    Definition transition (input : States * Symbols) : States * Symbols * Move :=
79      let (q, s) := input in
80      match q, s with
81      | a, false    => (b, true, R)
82      | b, false    => (c, false, R)
83      | c, false    => (c, true, L)
84      | a, true     => (halt, true, R)
85      | b, true     => (b, true, R)
86      | c, true     => (a, true, L)
87      | halt, false => (halt, false, C)
88      | halt, true  => (halt, true, C)
89      end.
90
91    (* If we are in the halting state, halt *)
92    Definition final (inputState : States) :=
93      match inputState with
94      | halt => true
95      | _ => false
96      end.
```

```
97
98    (* Run the busy beaver function *)
99    Fixpoint busy
100             (left : CoList Symbols)
101             (right : CoList Symbols)
102             (q : States) : States :=
103       let _ :=
104           (if final q
105            then q
106            else match TM States Symbols transition left right q with
107                 | Some (q', left', right') =>
108                   busy left' right' q'
109                 | None => halt
110                 end)
111       in q.
112
113   (* Tape input of infinite falses *)
114   CoFixpoint InfiniteFalse := CONS false InfiniteFalse.
115
116   Eval compute in (busy NIL InfiniteFalse a).
117
118 End BusyBeaver.
```

Bibliography

[1] Angiuli, Carlo, Hou, Kuen-Bang, Harper, and Robert. Computational Higher Type Theory III: Univalent Universes and Exact Equality, Dec 2017.

[2] Edwin Brady. Programming and Reasoning with Algebraic Effects and Dependent Types. *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming - ICFP 1*, 2013.

[3] Class and Instances Declarations. `https://downloads.haskell.org/~ghc/7.0.1/docs/html/users_guide/type-class-extensions.html`.

[4] Coq Implementation of a Turing Machine. `https://gist.github.com/casperbp/8c075a83d7402ff524fe8920eac93ad2`.

[5] Martín H. Escardó. A Self-Contained, Brief and Complete Formulation of Voevodsky's Univalence Axiom, 2018. arXiv:1803.02294.

[6] Institute for Advanced Study. *Homotopy Type Theory*. Institute for Advanced Study, 2013.

[7] Daniel R. Grayson. An Introduction to Univalent Foundations for Mathematicians. *Bulletin of the American Mathematical Society*, page 127, 2018.

[8] HoTT in Coq. `https://github.com/HoTT/HoTT`.

[9] Simon L. Peyton Jones and Philip Wadler. Imperative Functional Programming. *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL 93*, 1993.

[10] Per Martin-Löf. An Intuitionistic Theory of Types: Predicative Part. In H.E. Rose and J.C. Shepherdson, editors, *Logic Colloquium '73*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73 – 118. Elsevier, 1975.

[11] Eugenio Moggi. Notions of Computation and Monads. *Information and Computation*, 93(1):5592, 1991.

[12] Bertrand Russell. *The Principles of Mathematics*. University Press, 1903.

[13] Michael Sipser. Introduction to the Theory of Computation, Second Edition, 2005.

[14] The Coq Reference Manual. `https://coq.inria.fr/distrib/current/refman/`.

[15] Polina Vinogradova, Amy P. Felty, and Philip Scott. Formalizing Abstract Computability: Turing Categories in Coq. *Electronic Notes in Theoretical Computer Science*, 338:203 – 218, 2018. The 12th Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2017).

[16] Philip Wadler. Monads for Functional programming. *Advanced Functional Programming Lecture Notes in Computer Science*, page 2452, 1995.