Stella Lau

Theory and implementation of a general framework for big operators in Agda

Computer Science Tripos – Part II

Trinity College

18 May 2017

Proforma

Name	Stella Lau
College	Trinity College
Project Title	Theory and implementation of a general framework for big operators in Agda
Examination	Computer Science Tripos, Part II (June 2017)
Word Count	11987[1]
Project Originator	Timothy Griffin
Project Supervisors	Marcelo Fiore and Timothy Griffin

Original aims of the project

The project aims to *formalize* the notion of big operators, develop a general framework to *derive equations* manipulating big operator expressions, and provide a *formal implementation* in the dependently typed language and proof assistant Agda. The project aims to generalise previous work on big operators and develop a theoretical framework to derive more classes of equations in a more general way, providing a big operator library in Agda for formal verification of correctness as well as to demonstrate the *usability* of the approach.

Work completed

I have met all my proposed success criteria and have implemented two extensions. More importantly, I have developed the theory for a general framework for big operators based on the idea of monads and their algebras from the mathematical theory of categories. I have shown how different big operator equations can be derived in a general way using this framework, and provided an associated implementation of a big operator library in Agda that formalizes the theory. Furthermore, I have provided examples of proofs written using the library to demonstrate the usability of the approach.

[1] Number of words in the main body of the report as computed by TEXcount, `http://app.uio.no/ifi/texcount/`. This word count includes footnotes, but omits code listings and appendices.

Special difficulties

None.

Declaration of Originality

I, Stella Lau of Trinity College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed: Stella Lau

Date: 18 May 2017

Contents

Chapter 1

Introduction

Contents

In mathematics, it is often convenient to notationally express the iteration of a binary operator over a collection of values using a "big operator". As an example, $\sum$ is a big operator denoting the iteration of the binary operator $+$ over a collection of expressions, bracketed in some prescribed way (that does not matter as the operator is associative). Used in an expression such as

$$\sum_{k \in \{0,\ldots,N\}} \frac{x^k}{k!}$$

the big operator concisely and declaratively expresses the sum of the numbers $x^k/k!$ for all values of k from 0 to N. This notation reveals hidden symmetries and methods of manipulating and reasoning about the expression using a rich set of partitioning, reindexing, and commutation equations commonplace in the mathematical vernacular; see, for example, Table 1.1. These tools are ubiquitous in informal mathematics, but are not fully supported as formal constructs in their full generality in many proof assistants [6]. This dissertation contributes in this direction.

One can think of big operators as lifting a binary operator from acting on two values to acting on a finite collection of values. We generate a collection of values (for example numbers from 0 to N), apply some function to these values, and then operate on them with the big operator. We can formalize this idea to show that big operators share a common structure, and thus we should be able to derive equations

Equation	
Unit identity	$\bigvee_{x\in\{0\}} f(x) = f(0)$
Composition	$\sum_{x\in S} f(g(x)) = \sum_{y\in\{g(x)\mid x\in S\}} f(y)$
μ-identity	$\min_{x\in(l_1++\cdots++l_n)} f(x) = \min_{l\in[l_1,\ldots,l_n]} \min_{x\in l} f(x)$
Distributive law	$c\cdot\sum_{x\in t} f(x) = \sum_{x\in t}(c\cdot f(x))$
Homomorphism	$c^{\sum_{k\in K} f(k)} = \prod_{k\in K} c^{f(k)}$
Commutative law	$\prod_{x}\prod_{y} f(x,y) = \prod_{y}\prod_{x} f(x,y)$
Partition law	$\sum_{i=1}^{n} f(i) = \sum_{\substack{1\le i\le n\\ i\text{ odd}}} f(i) + \sum_{\substack{1\le i\le n\\ i\text{ even}}} f(i)$
Product law	$\sum_{i=1}^{5n} f(i) = \sum_{i=0}^{n-1}\sum_{j=1}^{5} f(5\cdot i+j)$
Commutativity	$\sum_{1\le k\le n} f(k) = \sum_{n\ge k\ge 1} f(k)$
Distributive product	$\sum_{j\in J,k\in K} f(j)g(k) = \left(\sum_{j\in J} f(j)\right)\left(\sum_{k\in K} g(k)\right)$
Filtering lemma	$\sum_{\substack{1\le k\le n\\ k\text{ even}}} \lfloor k/2\rfloor = \sum_{\substack{1\le k\le n\\ k\text{ even}}} k/2$

Table 1.1: Examples of big operator equations

to reason about them with minimal assumptions in a unified fashion. Furthermore, we can lift the particular properties (such as associativity or commutativity) of the binary operator to the big operator to obtain a rich set of equations to reason about the lifted operator.

This dissertation presents the theory and formalization of a new, general framework for big operators. From a theoretical point of view, it is based on the idea of monads and their algebras from the mathematical theory of categories. From a practical point of view, we formalize the theory through implementing it in Agda [28], a dependently typed programming language and proof assistant, and present a working big operator library.

1.1 Motivation and related work

This project is motivated and guided by the following.

Problem I. Formalization of big operators We aim to formally define the notion of a big operator and develop an associated framework for them, exploiting a common construction shared by big operators in order to derive equations to reason about big operators in a general form.

This goal was inspired by work from Bertot et al. [6], which presented an approach to big operators in terms of folding over sequences. Marcelo Fiore and Timothy Griffin [10] observed that this idea could be generalised to arbitrary containers and proposed a definition and thus approach for big operators in terms of monads and their algebras from category theory.

The challenge for this dissertation was to develop a mathematical theory for big operators based on the proposed definition from category theory, and attempt to derive equations such as those in Table 1.1 in the general framework.

Intuitively, big operator expressions encompass four aspects:

- A collection of values.
- A mapping from values in the collection to some other values.
- The type of the values we are computing.
- The big operator itself, which denotes how to combine the collection of values into a single value.

For example, in the expression

$$\sum_{x=1}^{N} x^2$$

we can think of the collection of values as the list $[1, 2, \dots, N]$ and the mapping as the function $f : \mathbb{N} \to \mathbb{N}$ given by $f(x) = x^2$. The type of the values we are computing is the natural numbers $\mathbb{N}$ and the big operator corresponds to summation over a list.

We will show that the collection of values and the action of mapping over a collection can be represented using the *monad* structure from category theory (also used in functional programming), and the big operator action of ranging over a collection (or monad) by an *algebra* for the monad.

Problem II. Manipulation of big operator expressions Big operators give rise to operations that let us manipulate expressions succinctly. We can rearrange, partition, reindex, and distribute over expressions in a way that reveals hidden symmetries [6]. For example, exp$(x + y) \equiv$ exp$(x) \times$ exp(y) expresses that exponentiation forms a homomorphism[1] between addition and multiplication. We can extend this to an associated big operator equation

$$c^{\sum_{a \in A} f(a)} \equiv \prod_{a \in A} c^{f(a)}$$

This is similar to the equation

$$\neg \left(\bigvee_{a \in A} f(a) \right) \equiv \bigwedge_{a \in A} \neg f(a)$$

in which $\neg$ is the homomorphism between $\vee$ and $\wedge$ given by De Morgan's law [14]. From these observations, one can synthesize a general big operator equation that states that if g is a homomorphism from $\oplus$ to $\otimes$, then

$$g \left(\bigoplus_{x \in A} f(x) \right) \equiv \bigotimes_{x \in A} g\,(f(x)) \tag{1.1}$$

We will derive such equations using minimal assumptions in a general form, finding it convenient to use the language of category theory. The abstract representation of big operators in terms of monads and their algebras gives rise to many equations just by the algebraic properties of the structures. By progressively instantiating the structures, we can derive more specific equations in a unified way.

[1] Recall that a map $g : A \mapsto B$ is a homomorphism for $\oplus$ on A and $\otimes$ on B if $\forall x, y \in A.\ g(x \oplus y) \equiv g(x) \otimes g(y)$.

Problem III. Formal implementation We aim to provide a computer implementation of our theoretical framework for big operators in a proof assistant as a proof of concept and as a way of formalizing the theory.

We will structure a modular implementation of big operators based on our framework in Agda. Agda's type theory is used to enforce correctness and ensure the careful handling of issues that are often ignored in informal mathematics but are crucial for formal verification.

Proof assistants such as Isabelle [27], HOL [16], Coq [39], and Agda [28] simplify the development of formal proofs. Providing support for big operators in proof assistants increases the classes of proofs that can be naturally expressed. Implementations of big operators in Isabelle and HOL are based on classical logic, which does not allow for direct program execution in the way that constructive logic does.

Our implementation is more similar to those in Coq and Agda, which are in the type-theory family of proof assistants and feature dependent types that can be used with the Curry-Howard correspondence (recalled in § 2.3.3) to create mechanically verified mathematical proofs. Our implementation was inspired by work by Bertot et al. [6] in Coq based on folding over sequences, a Part III dissertation by Leonhard Mackert [23] in Agda based on folding over lists, and Gustafsson and Pouillard [14, 15] in Agda based on "exploration functions."

Problem IV. Usability We would like our implementation to be practically usable and for proof-writing to resemble pen-and-paper mathematics. We provide an interface that abstracts away from the categorical foundations of the library and exploits Agda's *equational reasoning* (see § 2.3.5) to ensure that proof-writing is similar to pen-and-paper mathematics.

1.2 Contributions

This project addressed each of the problems presented in § 1.1 and contributed the following.

> **Problem I. Formalization of big operators**
> We developed and formalized a general framework for big operators based on monads and their algebras [10]. This has not been done at the abstract level of category theory.
>
> **Problem II. Manipulation of big operator expressions**
> We developed a theory for operator properties such as associativity,

distributivity, and commutativity in our categorical framework, and used it to derive general equations to manipulate big operator expressions.

Problem III. Formal implementation
We designed and implemented a modular, generic framework for big operators in Agda. We implemented and instantiated a variety of monads (tree, list, multiset, powerset) and operator properties. We present a big operator library with lemmas and big operator equations for common algebraic structures.

Problem IV. Usability
We developed a library that is based on the abstract categorical definitions while also providing an interface for users that requires no familiarity with category theory. The library exploits Agda's equational reasoning to enhance applicability, ensuring that proof writing is similar to pen-and-paper mathematics. We demonstrated the use of the library by writing proofs using monads with a variety of properties.

Table 3.3 on page 58 is a general version of Table 1.1, presenting examples of big operator equations that we derive generically in our categorical framework and implement in Agda.

Chapter 2

Preparation

Contents

This chapter begins with an informal analysis of big operators to motivate the remaining sections, followed by an introduction to the category theory, dependent type theory, and Agda fundamentals needed to understand the rest of the report.

2.1 Requirements analysis

Problem I. Formalization of big operators

Consider the following formula:

$$\sum_{0 \leq k \leq n} (a + bk) \tag{2.1}$$

From the above formula and equations in Table 1.1, we observe that big operator expressions have a *container* of values ($0, 1, \ldots, n$ in the example) and a *function* f that transforms the values in the container. In the example, $f(k) = a + bk$. Thus, the above can be expressed in a more general form:

$$\sum_{k\ in\ S} f(k)$$

In order to formalize the notion of big operators, we need to formalize such containers and functions acting upon them. In type theory, we can represent a collection of values of type A as a new type $T(A)$, where T is a type constructor. For instance, we might regard the collection of values to be of type *List* $\mathbb{N}$[1]. *List* acts as a type constructor that takes a type A and forms a type *List* A, the type of lists containing values of type A. Now given a mapping f from A to some type B, we need a way of lifting f to operate on a list of type A to compute a list of type B, that is to take a list $[0, 1, \ldots, n]$ and compute $[f(0), f(1), \ldots, f(n)]$.

A *monad* from category theory contains a type constructor T and a way of "lifting" maps $f : A \to B$ to maps from $T(A)$ to $T(B)$ (along with some desirable properties), which is precisely what we need. Thus, we represent the collection of values as relating to a monad (in our example, the list monad). Finally, we need a way of ranging over the container to transform our list of type *List* $\mathbb{N}$ (after applying f) to a single value of type $\mathbb{N}$; we need a map *List* $\mathbb{N} \to \mathbb{N}$. This corresponds to an *algebra for the list monad*, which consists of a type B and a map *List* $B \to B$ along with some desirable properties.

We find it convenient to use the language of category theory, which provides a general theory of functions as well as a rich conceptual background for reasoning about structures and their "admissible transformations". § 2.2 provides the necessary category theory background to understand the report.

[1] We will show that a more appropriate representation is a multiset.

Problem II. Manipulation of big operator expressions

We want equations and lemmas to reason about big operator expressions. For example, the following equation holds by commutativity of addition, which implies that summing over the list $[0, 1, \dots, n]$ is equivalent to summing over $[n, n-1, \dots, 0]$.

$$\sum_{0 \le k \le n} (a + bk) \equiv \sum_{n \ge k \ge 0} (a + bk) \tag{2.2}$$

By representing the container with the *List* monad, we "lose" the commutativity property of addition as $[0, 1, \dots, n]$ and $[n, n-1, \dots, 0]$ are not equivalent lists. However, the multisets[2] $\{[0, 1, \dots, n]\}$ and $\{[n, n-1, \dots, 0]\}$ are equivalent, and thus the multiset is the desired monad. In category theory, we say that the multiset monad is a *commutative monad*, and we can derive equations for big operators acting on any commutative monad. We will show that container types that big operators act upon (e.g. trees, lists, multisets, and sets) correspond to properties of the underlying algebraic structure associated with the binary operator (i.e. lists to monoids and multisets to commutative monoids).

Equation (2.2) can be represented more generally as follows:

$$S \equiv_{TX} S' \Rightarrow \sum_{k \ in \ S} f(k) \equiv \sum_{k \ in \ S'} f(k) \tag{2.3}$$

Equality is informally defined for now, but this essentially states that if a container S is equivalent (in this case, multiset equivalent) to a container S', then operating over S is equivalent to operating over S'.

Another common operation is index renaming:

$$\sum_{n \ge k \ge 0} (a + bk) \equiv \sum_{0 \le k \le n} (a + b(n - k)) \tag{2.4}$$

We can write $\{[n, n-1, \dots, 0]\}$ as $\{[g(0), g(1), \dots, g(n)]\}$, where $g(k) = n - k$, and represent the above equation more generally (see § 3.1.3)):

$$\sum_{k \ in \ T(g)S} f(k) \equiv \sum_{k' \ in \ S} f(g(k')) \tag{2.5}$$

This equation states that operating over the container S after transformation with a lifted function g is equivalent to composing f with g and operating over S. Note that eqs. (2.3) and (2.5) are general equations that do not make any assumptions about the container (monad) or operator (algebra); in other words, they do not assume associativity, commutativity, or any other property.

[2] A multiset is a set-like object in which ordering is irrelevant but multiplicity is explicitly significant. For example, $\{a, a, b\}$ and $\{a, b\}$ are equal sets but are not equal multisets.

Problem III. Formal implementation

We will implement our big operator framework in Agda's type theory (introduced in §§ 2.3.1 and 2.3.2), which will be used for formal verification of the theory. We will formalize the notion of equality (§ 2.3.4). By the Curry-Howard correspondence (recalled in § 2.3.3), Agda can be used as a proof assistant. We will produce a working big operator library.

Problem IV. Usability

Agda's equational reasoning will be used to enhance applicability and ensure that proof writing is similar to pen-and-paper mathematics. § 2.3.5 provides an example of a proof written using equational reasoning in Agda.

2.2 Category theory background

Category theory was invented as a way of studying and characterizing different kinds of mathematical structures in terms of their "admissible transformations" [2]. The basic structure of category theory is a category, which consists of objects $A, B, ...$ and arrows between objects $f : A \rightarrow B$ along with two function-like properties: the ability to compose arrows associatively and the existence of an identity arrow for each object.

This notation is familiar from set theory and type theory, in which the objects $A, B, ...$ correspond to *types* and arrows $f : A \rightarrow B$ to *functions* between types. Category theory is built upon objects and arrows and generalizes the idea of sets and functions by saying that there are many kinds of structures other than sets and many kinds of mappings between structures other than functions. For example, sets are characterized by membership whereas an object in a category does not necessarily have an associated notion of membership.

Although the initial motivation for category theory was not directly related to type theory, a number of correspondences have been found between particular kinds of categories and type theories. For example, Lambek showed that a particular type of category, called a Cartesian-closed category, corresponds to the typed λ-calculus [22] and Seely showed that locally Cartesian-closed categories correspond to Martin-Löf type theory [37], upon which Agda was based. This interplay between category theory and type theory, known as *categorical logic*, provides a rich mathematical background and categorical framework for type-theoretic constructions.

2.2.1 Category

Definition (Category)**.** A *category* $\mathcal{C}$ consists of *objects* $A, B, C, ...$ and *arrows* $f, g, h, ...$ such that

- For each arrow f there are given objects $\mathrm{dom}(f)$, $\mathrm{cod}(f)$ called the *domain* and *codomain* of f. We write $f : A \to B$ to indicate that $A = \mathrm{dom}(f)$ and $B = \mathrm{cod}(f)$.
- Given arrows $f : A \to B$ and $g : B \to C$, there is an arrow $g \circ f : A \to C$ called the *composite* of f and g.
- For each object A there is given an arrow $\mathbf{1}_A : A \to A$ called the *identity arrow* of A.
- For all arrows $f : A \to B,\ g : B \to C,\ h : C \to D$, the following holds:

$$h \circ (g \circ f) = (h \circ g) \circ f \qquad \text{(associativity law)} \qquad (2.6)$$

- For all $f : A \to B$, the following holds:

$$f \circ \mathbf{1}_A = f = \mathbf{1}_B \circ f \qquad \text{(unit laws)} \qquad (2.7)$$

In type theory, we can think of the objects A, B, C ... as types and the arrows as functions between types. A familiar example of a category is the category $\mathcal{S}et$, in which the objects are sets and the arrows are functions. Note that category theory itself does not insist that objects have elements.

2.2.2 Functor

Type constructors in functional programming and type theory like ×, →, and *List* correspond to *functors* in category theory. Functors are maps between categories. A functor $F : \mathcal{C} \to \mathcal{D}$ from category $\mathcal{C}$ to $\mathcal{D}$ maps objects of $\mathcal{C}$ to objects of $\mathcal{D}$ and arrows of $\mathcal{C}$ to arrows of $\mathcal{D}$ such that the source and target of the arrow are respected. That is, if F maps objects A and B in $\mathcal{C}$ to objects FA and FB in $\mathcal{D}$, then it should map arrow $f : A \to B$ in $\mathcal{C}$ to an arrow $Ff : FA \to FB$ in $\mathcal{D}$. F preserves identity and composition.

Definition (Functor)**.** A *functor* $F : \mathcal{C} \to \mathcal{D}$ between categories $\mathcal{C}$ and $\mathcal{D}$ is a mapping of objects to objects and arrows to arrows such that

$$F(f) : F(A) \to F(B) \qquad \text{for } f : A \to B \tag{2.8}$$

$$F(g \circ f) = F(g) \circ F(f) \qquad \text{for } f : A \to B \text{ and } g : B \to C \tag{2.9}$$

$$F(\mathbf{1}_A) = \mathbf{1}_{F(A)} \tag{2.10}$$

In the context of $\mathcal{S}et$ and type theory, a functor is both a type constructor (mapping types to types) and a map from functions between types to functions between types. Note that a functor consists of two maps, one on objects and one on morphisms. We overload the symbol F.

An example of a functor is **List** : $\mathcal{S}et \to \mathcal{S}et$. The object map of the functor maps a set (or type) A to the set (or type) of lists over A (finite sequences of the form $[a_1, \dots, a_n]$ where each a_i is an element of A). The morphism map of the functor maps a function $f : A \to B$ to the function commonly known as *map* f, mapping list $[a_1, \dots, a_n]$ to $[f(a_1), \dots, f(a_n)]$. This is written **List** f : **List** $A \to$ **List** B

2.2.3 Natural transformation

Natural transformations are maps between functors and can be used to represent polymorphic functions. Polymorphic functions like *reverse*$_A$: **List** $A \to$ **List** A and *map*$_{A,B}$: $(A \to B) \to$ (**List** $A \to$ **List** B) represent *families* of functions generic in the type parameters. If **List** is a functor from $\mathcal{C} \to \mathcal{C}$, then *reverse* is a map from **List** $\to$ **List**. We want polymorphic functions to "act the same way" on different types. This is "naturality" in category theory, which is intuitively a condition expressing that natural transformations are structure-preserving maps. Reverse is natural if for every arrow $f : A \to A'$ in $\mathcal{C}$, the following diagram commutes:

$$\begin{tikzcd}
\textbf{List } A \arrow[r, "\textbf{List } f"] \arrow[d, "reverse_A"'] & \textbf{List } A' \arrow[d, "reverse_{A'}"] \\
\textbf{List } A \arrow[r, "\textbf{List } f"'] & \textbf{List } A'
\end{tikzcd}$$

That is, one may reverse a list before or after mapping a function over it without affecting the result.

Definition (Natural transformation)**.** Let $\mathcal{A}$ and $\mathcal{B}$ be categories and let $\mathcal{A} \overset{F}{\underset{G}{\rightrightarrows}} \mathcal{B}$ be two functors. A *natural transformation* $\alpha : F \to G$ is a family

$\left(F(A) \xrightarrow{\alpha_A} G(A)\right)_{A \in \mathcal{A}}$ of maps in $\mathcal{B}$ such that for every map $A \xrightarrow{f} A'$ in $\mathcal{A}$, the square

$$\begin{tikzcd}
FA \arrow[r, "F(f)"] \arrow[d, "\alpha_A"'] & FA' \arrow[d, "\alpha_{A'}"] \\
GA \arrow[r, "G(f)"'] & GA'
\end{tikzcd} \tag{2.11}$$

commutes in $\mathcal{B}$.

2.2.4 Monad

The theory of monads was initially developed to express certain aspects of algebraic geometry. Monads are used in functional programming to express types of computational effects. For example in Haskell [29], the IO system is constructed using a monad. In Haskell, the `Monad` class defines two basic operations: »= (*bind*) and *return*.

```
class Monad m where
  (>>=)     :: m a -> (a -> m b) -> m b
  (>>)      :: m a -> m b -> m b
  return    :: a -> m a
  fail      :: String -> m a
```

Subject to a set of laws, the *return* operation injects a value into a monadic type and the *bind* operation composes operations on monadic values. The bind operation takes as arguments a monadic value of type `m a` and a function (`a` → `m b`) and, intuitively, "unwraps" the value of type `a` embedded in the input monadic value and passes it as input to the function, which creates a new monadic value of type `m b`.

This definition of monad is equivalent to the category theoretic definition below (proved in Appendix A).

Definition (Monad)**.** A *monad* on a category $\mathcal{C}$ consists of an endofunctor[3] $T : \mathcal{C} \to \mathcal{C}$ and two natural transformations $\eta : \mathbf{1}_{\mathcal{C}} \to T$ and $\mu : T^2 \to T$ sat-

[3] An endofunctor is a functor that maps a category to itself.

isfying the following two commutative diagrams.

$$\begin{tikzcd}
T \arrow[r, "\eta_T"] \arrow[dr, "\mathbf{1}_T"'] & T^2 \arrow[d, "\mu"] & T \arrow[l, "T\eta"'] \arrow[dl, "\mathbf{1}_T"] \\
& T &
\end{tikzcd} \tag{2.12}$$

$$\begin{tikzcd}
T^3 \arrow[r, "\mu_T"] \arrow[d, "T\mu"'] & T^2 \arrow[d, "\mu"] \\
T^2 \arrow[r, "\mu"'] & T
\end{tikzcd} \tag{2.13}$$

That is,

$$\mu \circ \eta_T = \mathbf{1} = \mu \circ T\mu \tag{2.14}$$

$$\mu \circ \mu_T = \mu \circ T\mu \tag{2.15}$$

2.2.5 T-algebra

Definition (T-algebra)**.** Assume that (T, η, μ) is a monad on a category $\mathcal{C}$. A T-algebra (A, α) is an object A of $\mathcal{C}$ together with an arrow $\alpha : TA \to A$ of $\mathcal{C}$ such that

$$\mathbf{1}_A = \alpha \circ \eta_A \tag{2.16}$$

$$\alpha \circ \mu_A = \alpha \circ T\alpha. \tag{2.17}$$

Example (μ-algebra)**.** For every monad (T, η, μ) on a category $\mathcal{C}$ and every object X of $\mathcal{C}$, we can define a T-algebra $(TX,\ \mu_X : TTX \to TX)$. For this to be a T-algebra, the following diagrams corresponding to eqs. (2.16) and (2.17) (replacing A with TX and α with μ_X) must commute.

$$\begin{tikzcd}
TX \arrow[r, "\eta_{TX}"] \arrow[dr, "\mathbf{1}_{TX}"'] & TTX \arrow[d, "\mu_X"] \\
& TX
\end{tikzcd}
\qquad
\begin{tikzcd}
T^3X \arrow[r, "\mu_{TX}"] \arrow[d, "T\mu_X"'] & T^2X \arrow[d, "\mu_X"] \\
T^2X \arrow[r, "\mu_X"'] & TX
\end{tikzcd}$$

These diagrams correspond to eqs. (2.12) and (2.13) in the definition of a monad. So (TX, μ_X) is a T-algebra, which we refer to as the μ-algebra.

2.3 Agda

The relationship between Cartesian-closed categories and typed lambda calculus can be extended via the Curry-Howard-Lambek correspondence to a three-way correspondence including intuitionistic logic [22, 38]. Intuitionistic type theory was introduced by Martin-Löf based on intuitionistic logic as an alternative foundation of mathematics. Martin-Löf showed [24] that the additional richness of type theory, as compared with first-order logic, makes it usable as a programming language. This is the basis of Agda, a dependently-typed functional language and proof assistant that uses intuitionistic type theory as the mathematical foundation for formal proofs.

Unlike in Hindley-Milner style languages like ML where there is a separation between types and values, in dependently-typed languages like Agda, types can depend on values and can appear as arguments and results of functions [28]. Curry and Howard discovered a correspondence between logic and type theory and showed that, with the introduction of dependent types, every proof in predicate logic can be represented as a term of a suitable typed lambda calculus [42]. As a result of this Curry-Howard correspondence, which, informally, says that for each logical *proposition* there is a *type* and for each *proof* of a given proposition there is a *program* of the corresponding type [38], dependent type systems have been used as the basis of proof assistants.

This section introduces Agda, dependent type theory, and the Curry-Howard correspondence.

2.3.1 Basic data types and pattern matching

Like in ML, pattern matching over algebraic data types (ADTs) is an important construct in Agda. Data types are declared with the data declaration, which gives the name and type of the data type along with the constructors and their types. For example, the type Bool contains two constructors, true and false.

```
data Bool : Set where
  true  : Bool
  false : Bool
```

The type of Bool is Set, the type of small types. Roughly, this is like the "class of sets”[4].

We can define functions over Bool by pattern matching:

```
not : Bool → Bool
not true = false
not false = true
```

The type checker checks that the function covers all possible cases, as Agda functions are total (and are not allowed to crash).

Like in ML, data types can be parametrised by other types. The type of lists of elements of an arbitrary type is defined by

```
infixr 5 _::_
data List {ℓ} (A : Set ℓ) : Set ℓ where
  []   : List A
  _::_ : A → List A → List A
```

Agda supports mixfix operators, with the underscores around ‘::’ indicating the location of the arguments. The cons operator ‘::’ takes an element of type A and an element of type List A and constructs an element of type List A. Here ℓ is a universe level, with the curly braces indicating that it is implicit and does not have to be specified by the user if it can be inferred (note that type inference in Agda is undecidable).

2.3.2 Dependent types

A dependent type is a type whose definition depends on a value. Informally, dependent types are similar to indexed families of sets[5]. Dependent types can be used to represent *dependent functions* such as polymorphic functions, which are a family of functions indexed by type. Indexing on parametrisation can be represented via *dependent pairs*, in which the second value may depend on the first (for instance, in existential quantification).

More formally, given a set A and a family of sets B_a indexed by elements a of A,

[4] There is a hierarchy of increasingly large types. The type of Set is Set1, whose type is Set2, etc. These universe levels will be present in the code for generality, but can be safely ignored by the reader.

[5] We write $\{B_a \mid a \in A\}$ to denote a family of sets B_a indexed by elements $a \in A$, where A is a set.

we get a set

$$\prod_{a \in A} B_a \overset{def}{=} \left\{ F \in \left(A \to \bigcup_{a \in A} B_a \right) \mid \forall (a, b) \in F.\, b \in B_a \right\}$$

of *dependent functions*. Each $F \in \prod_{a \in A} B_a$ is a function that associates to each $a \in A$ an element in B_a (written $F\ a$) [32]. In Agda, we write $(a : A) \to B$ for the dependent function that takes an element of type A and returns an element of a type B instantiated at a. For example, we can define the append and map functions for arbitrary types as follows:

```
_++_ : ∀ {ℓ} {A : Set ℓ} → List A → List A → List A
[]  ++ ys = ys
( x ∷ xs) ++ ys = x ∷ (xs ++ ys)

map : ∀ {ℓ k} {A : Set ℓ} {B : Set k} → (A → B) → List A → List B
map f []  = []
map f (x ∷ xs) = f x ∷ map f xs
```

Dependent types can be used to define the type of lists of a given length:

```
data Vec {ℓ} (A : Set ℓ) : ℕ → Set a where
  [] : Vec A zero
  _∷_ : ∀ {n} (a : A) (as : Vec A n) → Vec A (suc n)
```

The type of Vec A is $\mathbb{N} \to$ Set, so Vec A is a family of types indexed by natural numbers. Agda guarantees that for a vector v : Vec A k, not only is every element of v of type A but, moreover, that v has exactly k elements. This is unlike ML, for instance.

2.3.3 Curry-Howard correspondence

Classically, a proposition is just a truth value, an element in the set $\{$*true, false*$\}$. In intuitionistic logic,

> a proposition is defined by laying down what counts as a proof,

and

> a proposition is true if it has a proof [34].

So in intuitionistic logic, truth is identified with provability. If we consider the idea that a proposition is defined by stating how its proofs are formed (for instance

Logic		$\Longleftrightarrow$		Type theory
True formula	T	$\Longleftrightarrow$	$\top$	Unit type
False formula	F	$\Longleftrightarrow$	$\bot$	Empty type
Conjunction	$A \wedge B$	$\Longleftrightarrow$	$A \times B$	Product type
Disjunction	$A \vee B$	$\Longleftrightarrow$	$A \uplus B$	Sum type
Implication	$A \Rightarrow B$	$\Longleftrightarrow$	$A \rightarrow B$	Function type
Universal quantification	$\forall a \in A.\ B(a)$	$\Longleftrightarrow$	$\Pi\ (a : A)\ B(a)$	Dependent function
Existential quantification	$\exists a \in A.\ B(a)$	$\Longleftrightarrow$	$\Sigma\ (a : A)\ B(a)$	Dependent pair
Modus ponens		$\Longleftrightarrow$		Function application
Provability		$\Longleftrightarrow$		Inhabitation

Table 2.1: Curry-Howard correspondence for intuitionistic logic

a proof of $A \wedge B$ has the form (a, b) where a is a proof of A and b is a proof of B) and that a set is defined by prescribing how its elements are formed, then there is a correspondence between the notions of propositions and sets and the associated notions of a proof of a proposition and an element of a set. This isomorphism is the formulae-as-types interpretation of the Curry-Howard correspondence, presented in Table 2.1.

The type theory equivalents of truth ($\top$) and falsity ($\bot$) in Agda are given below:

```
data ⊤ : Set where
  tt : ⊤

data ⊥ : Set where
```

The type $\bot$ has no constructor, which means it is impossible to construct values of type $\bot$. This is consistent with the idea that you can not prove false (you can not inhabit the $\bot$ type).

Product types (corresponding to conjunction) can be constructed using the record keyword. Records group terms together and are essentially tuples of named fields. The optional constructor introduces syntax to construct terms of the record type.

```
record _×_ {ℓ} {k} (A : Set ℓ) (B : Set k) : Set (ℓ ⊔ k) where
  constructor _,_
  field
    proj₁ : A
    proj₂ : B
```

A proof of $A \vee B$ is either inj$_1$ a, where a is a proof of A, or inj$_2$ b, where b is an element of type B.

```
data _⊎_ {ℓ k} (A : Set a) (B : Set k) : Set (ℓ ⊔ k) where
  inj₁ : (a : A) → A ⊎ B
  inj₂ : (b : B) → A ⊎ B
```

We can prove that disjunction is commutative $A \vee B \to B \vee A$ by pattern matching on the element of type $A \uplus B$ and swapping the constructor to build an element of type $B \uplus A$.

```
⊎—comm : ∀ {ℓ k} {A : Set ℓ} {B : Set k} → A ⊎ B → B ⊎ A
⊎—comm (inj₁ a) = inj₂ a
⊎—comm (inj₂ b) = inj₁ b
```

The type of ⊎—comm states that given a proof that $A \vee B$ is true, we can construct a proof that $B \vee A$ is true.

Dependent product and sum types can be used to express universal and existential quantification. Indexed products $\Pi\ (a : A)\ B(a)$ and indexed sums $\Sigma\ (a : A)\ B(a)$ correspond, respectively, to universal quantifications $\forall a \in A.\ B(a)$ and existential quantifications $\exists x \in A.\ B(x)$. A proof of $\forall a \in A.\ B(a)$ takes the form of a function $\lambda a \in A.\ b(a)$ where $b(a)$ is a proof of $B(a)$ provided a is a proof of A. This is the dependent function type presented in § 2.3.2.

Similarly, a proof of $\exists a \in A.\ B(a)$ takes the form of a pair $(a, b(a))$ where $b(a)$ is a proof of $B(a)$ provided a is a proof of A. The type of the dependent pair is as follows:

```
record ∑ {ℓ k} (A : Set ℓ) (B : A → Set k) : Set (ℓ ⊔ k) where
  field
    proj₁ : A
    proj₂ : B proj₁
```

Modus ponens states that if A and $A \Rightarrow B$ are true, B is true. In other words, given elements a and f of type A and $A \to B$, we need to construct an element of type B. This is given by function application in the Curry-Howard correspondence, as demonstrated below.

```
modus—ponens : ∀ {ℓ k} {A : Set ℓ} {B : Set k} → A → (A → B) → B
modus—ponens a f = f a
```

These examples illustrate how inhabitability in type theory corresponds to provability in logic.

2.3.4 Equality

In type theory, one distinguishes various notions of equality. These are *definitional equality*, which is Agda's internal notion of program equality, and *propositional equality*, which is a type-theoretic notion of equivalence.

Definitional equality is used by the type checker and is a meta-theoretic relation between terms and types. When defining a function such as

```
not : Bool → Bool
not true = false
not false = true
```

one adds the defining equation to Agda's definitional equality.

Crucially, note that in order to prove that two programs are equal, definitional equality can not be directly used. A proof is a program with a type, so the type in this case should express that two things are equal. For example, the functions $\lambda x.\, x + 1$ and $\lambda x.\, 1 + x$ are equal, but are not definitionally equal. In type theory, we introduce an identity type *Id* $A\ a\ b$ when A is a type and a and b are both equal terms of A. This is the "type of proofs that $a = b$". When *Id* $A\ a\ b$ is inhabited, we say that a is *propositionally equal* to b.

In contrast to ML, where one needs a distinction between types and *equality* types, every type in Agda has an identity type, including function types. Propositional equality is defined as follows in Agda:

```
data _≡_ {a} {A : Set a} (x : A) : A → Set a where
  refl : x ≡ x
```

Two elements of the same type are propositionally equal if we can find a proof or program that constructively shows that the two elements are equal. In defining a term of the $x \equiv y$ type, one provides a proof that x is propositionally equal to y.

For example, we prove that $2 + 2 \equiv 4$:

```
pf₁ : 2 + 2 ≡ 4
pf₁ = refl
```

Agda's type checker reduces both sides of the equality until they are equal. Note that definitional equality entails propositional equality, but not vice versa.

Martin-Löf type theories are often divided into two camps: *extensional* and *intensional*. Extensional theories identify definitional equality (used in type checking) with propositional equality. Intensional theories, like Agda's, distinguish between definitional and propositional equality. Propositional equality is undecidable, and identifying definitional equality with propositional equality makes type checking undecidable [18], an undesirable feature.

2.3.5 Proof example

We present a proof (or program) that list append is associative[6].

```
++-assoc : ∀ {a} {A : Set a} (xs ys zs : List A) →
   (xs ++ ys) ++ zs ≡ xs ++ (ys ++ zs)
```

We prove this by induction on the length of the first list. Recall that list append is defined as

```
_++_ : ∀ {a} {A : Set a} → List A → List A → List A
[]  ++ ys = ys
( x ∷ xs) ++ ys = x ∷ (xs ++ ys)
```

For the base case (*xs* being []), ([] ++ *ys*) ++ *zs* is definitionally equal to [] ++ (*ys* ++ *zs*), so the proof is refl:

```
++-assoc [] ys zs = refl
```

For the inductive step, we show that ((*x* ∷ *xs*) ++ *ys*) ++ *zs* is propositionally equal to (*x* ∷ *xs*) ++ (*ys* ++ *zs*) using Agda's *equational reasoning*, which provides pen-and-paper like syntax for writing proofs using the transitivity property of equality.

[6] Based on [1]

```
++-assoc (x ∷ xs) ys zs =
  begin
    ((x ∷ xs) ++ ys) ++ zs
  ≡⟨ refl ⟩
    x ∷ ((xs ++ ys) ++ zs)
  ≡⟨ cong (_∷_ x) (++-assoc xs ys zs) ⟩
    x ∷ (xs ++ (ys ++ zs))
  ≡⟨ refl ⟩
    (x ∷ xs) ++ (ys ++ zs)
  ■
```

((x ∷ xs) ++ ys) ++ zs is definitionally equal to x ∷ ((xs ++ ys) ++ zs) and as definitional equality entails propositional equality, the proof is given by refl. The next step uses congruence, which is a proof that if $x \equiv y$ then $f\ x \equiv f\ y$. In Agda:

```
cong : ∀ {a b} {A : Set a} {B : Set b}
  (f : A → B) {x y} → x ≡ y → f x ≡ f y
cong f refl = refl
```

Thus, cong applied to _∷_ x and, inductively, ++-assoc xs ys zs gives the proof that x ∷ ((xs ++ ys) ++ zs) is equal to x ∷ (xs ++ (ys ++ zs)).

2.4 Summary

This project has two components: (i) a formal *theory* of big operators and (ii) an *implementation* of a big operator library in Agda. Rather than developing the theory and implementation separately, we followed an iterative cycle of development: developing new theory then implementing it in Agda, which formalizes the theory and proves its correctness. We identified some *ubiquitous equations* in Table 1.1 to structure the development.

We identified an open-source Agda category theory library [31] that provided definitions for categories, functors, and natural transformations. This, along with the idea of defining big operators in terms of monad algebras[10] and Table 3.1 on page 32, was the starting point of the project. From here, the components break down as follows.

> **Big operators** Define big operators mathematically and in Agda. Decide what category to work with and develop and implement theory

for the category of choice. Prove and implement the generic equations in Table 3.1.

Monads: associativity, commutativity, idempotence Operators with different properties correspond to different monads. Implement the tree, list, multisets, and powerset monads in order to express equations such as $\sum_x \sum_y f(x, y) = \sum_y \sum_x f(x, y)$.

Algebras: distributivity and algebraic structures Develop and implement the correspondence between algebras and structures like monoids and semirings. Derive distributivity equations such as $c \cdot \sum_x f(x) = \sum_x c \cdot f(x)$.

Usability Ensure the library is usable without knowledge of category theory. Instantiate common big operators and provide lemmas for list manipulations such as filtering and partitioning to express equations such as

$$\sum_{0 \leq i \leq N} f(i) = \sum_{\substack{0 \leq i \leq N \\ i \text{ even}}} f(i) + \sum_{\substack{0 \leq i \leq N \\ i \text{ odd}}} f(i)$$

Examples Provide example proofs using the library and evaluate the difficulty and steps needed to go from an informal big operator equation to a concrete proof.

This chapter provided the necessary category theory and Agda background to understand the rest of this report. Next, we discuss the theory and implementation of the components above.

Chapter 3

Implementation

Contents

This chapter details the theory of a categorical framework for big operators based on monad algebras along with an associated implementation of a big operator library in Agda.

The work of this dissertation involved translating between and, in a sense, unifying different languages: (i) the language of *abstract algebra* when reasoning about familiar big operators, (ii) the language of *category theory*[1] when generalizing big operators and presenting a categorical framework, (iii) the language of *dependent type theory* when providing a more familiar syntax for the former, and (iv) the language of *Agda* when writing formal, type-checked proofs and implementing the library.

This chapter develops the components of the categorical framework in all four languages. Big operators are defined and implemented in § 3.1. § 3.2 and § 3.3 describe the interpretations of monads and algebras in the framework and their associated big operator properties. § 3.4 explains lemmas implemented to increase usability of the library.

3.1 Big operators

3.1.1 Mathematical definition of big operators

Informally, a big operator consists of four components:

1. A *container* of values of type X. We think of this as a value of type TX, where T is a functor in a monad (T, η, μ) that acts as a type constructor.

   For instance, with the list monad, the container is of type *List X*. Then $\eta_X : X \to$ *List X* injects the singleton $x \ : \ X$ into $[x]$, and $\mu_X :$ *List (List X)* $\to$ *List X* flattens a list of lists into a list by concatenating the lists.

2. A *type* A that is the type of the result.

3. A *function* $f : X \to A$ that maps each value in the container of type X to a value of type A. The morphism map of the functor T lifts the function to the monadic type (e.g. map for lists), with $T(f) : TX \to TA$.

4. A *reduction function* of type $TA \to A$ that reduces the values in the container. The type A and reduction function are combined into an algebra for

[1] More specifically, of locally Cartesian-closed categories

> the monad[2] $(A, \alpha : TA \to A)$. The algebra laws express correctness properties for the reduction. An example of α is the summation function, which takes a value of type *List* $\mathbb{N}$ and returns a value of type $\mathbb{N}$.

The above is deliberately imprecise to provide intuition: the languages of type theory and category theory have been mixed and discussion of the category upon which the monads and thus algebras were defined avoided. As per [19], a semantic function can be defined that interprets the syntax of dependent type theory in locally Cartesian-closed categories. Then types X, A, and TX are objects in the category of choice. The function $f : X \to A$ should be an arrow in the category for the expression $T(f)$ to make sense. As types are objects, it was implicitly assumed that objects have elements. The concept of "elements of" does not exist for arbitrary categories and suggests that objects in our category should be set-like.

Setoids A logical choice of a category with set-like objects is the category of $\mathcal{S}ets$, in which the objects are sets and the arrows functions. However, types and sets are not mathematically equivalent [5] and we need a way to represent equality, more specifically extensional equality, in Agda's intentional type theory (recall § 2.3.4).

Frequently in mathematics, when defining an equivalence relation on a set, we immediately form the quotient set that turns equivalence into equality. However, in type theories that do not have an established notion of quotient types such as Agda's, formalizing mathematics often requires representing sets as *setoids* [4]. These are sets (or types) together with an explicit equivalence relation representing the intended notion of equality between elements of the set.

Thus, rather than using $\mathcal{S}ets$, we choose to work with the category of $\mathcal{S}etoids$, a locally Cartesian-closed category that interacts well with dependent type theory [4, 19] (and has many desirable properties beyond the scope of this report). In $\mathcal{S}etoids$, the objects are setoids and an arrow $X \longrightarrow A$ is a function from X to A that preserves the equality of the setoid. That is, such that

$$\forall x_1, x_2 \in X.\ x_1 \equiv_X x_2 \Rightarrow f(x_1) \equiv_A f(x_2)$$

Note that we can define a setoid of functions based on extensional equality with the equivalence relation

$$f \equiv_{X \to A} g \iff \forall x \in X.\ f(x) \equiv_A g(x)$$

Thus, for any two objects X and A, the setoid of functions from X to A is an object in $\mathcal{S}etoids$. From now on, we place ourselves in the category of $\mathcal{S}etoids$. Thus, for

[2] As defined in § 2.2.5.

readability when we say "function", we mean "function that preserves equality" or "arrow in $\mathcal{S}etoids$". We use the notations $x \in X$ and $x : X$ interchangeably, depending on whether we are working with the language of mathematics built on set theory or on type theory, and often treat X more familiarly as a set rather than a setoid. The distinctions are made clear in the Agda implementation, which increases confidence that details have not been erroneously ignored.

Big operators Previous work [14] defined big operators as follows:

> A big operator for a small operator $\_ \oplus \_$ together with default value ϵ of type U is a function that, given a body of type $A \to U$ for some type A, will apply some values of A and combine them with the $\_ \oplus \_$ operator. If there are no values to apply, ϵ is returned.

We generalise and formalize this by defining big operators as follows:

Definition (Big operator)**.**

> A big operator for a given monad (T, η, μ) on the category $\mathcal{S}etoids$ and T-algebra (A, α) is a function (preserving equality) that, for every setoid X, given a function $f : X \to A$ preserving equality on X and a structured value t of type TX, applies the function $T(f)$ to t and combines the result with α. In mathematical form:
>
> $$\begin{aligned} @ &: \forall X. (X \to A) \to TX \to A \\ @\ X\ f\ t &\overset{\text{def}}{=} (\alpha \circ T(f))\ t \end{aligned} \tag{3.1}$$

Intuitively, given $f : X \to A$ and $t : TX$, applying $T(f) : TX \to TA$ gives a value of type TA. Composing the result with $\alpha : TA \to A$ returns a value of type A. In a sense, $@\ X$ is a functional taking an arrow $X \to A$ and returning another arrow $TX \to A$. Thus $@\ X$ is an arrow in $\mathcal{S}etoids$ from $X \to A$ to $TX \to A$.

We write the above expression using the notation below, where we treat X as an implicit argument and omit it.

$$\underset{x \in t}{@} f(x)$$

3.1.2 Implementation of big operators in Agda

In Agda, we use a category theory library [31] that provides fundamental category theory definitions to define big operators as follows:

```
module Bigop {c ℓ} (M : Monad (Setoids c ℓ)) (T : T−algebra M) where
  bigop : ∀ {X : Setoid c ℓ} → (f : X ⟶ A) → (t : Setoid.Carrier (F₀ X)) →
    Setoid.Carrier A
  bigop f t = (α ∘ F₁ f) ⟨$⟩ t
```

We defined a module parametrised by a monad on the category Setoids and a T−algebra on the monad, which gives us the setoid A and function $\alpha : \mathsf{F}_0\, X \longrightarrow \mathsf{A}$. F_0 and F_1 correspond to the object and morphism map of T respectively. The long arrow $\longrightarrow$ is a type constructor for the type of functions preserving equality.

The big operator takes (implicitly) a setoid *X*, a function preserving equality *f*, and an element of the carrier type of the setoid TX, written as Setoid.Carrier ($\mathsf{F}_0\, X$), and returns an element of the carrier type of setoid A.

3.1.3 Generic big operator equations

We can derive some general big operator equations in our framework without assuming anything about the monad or algebra. These are presented in Table 3.1.

$$\textcircled{\alpha}\ \mathrm{id}\ t = \alpha(t) \tag{3.2}$$

$$\textcircled{\alpha}\ f\ \eta_X(a) = f(a) \tag{3.3}$$

$$\textcircled{\alpha}\ (f \circ g)\ t = \textcircled{\alpha}\ f\ T(g)(t) \tag{3.4}$$

$$\textcircled{\alpha}\ f\ (\textcircled{\mu}\ g\ t) = \textcircled{\alpha}\ (\lambda\, x.\, \textcircled{\alpha}\ f\ g(x))\ t \tag{3.5}$$

$$\textcircled{\alpha}\ f\ \mu_X(s) = \underset{t \in s}{\textcircled{\alpha}}\ \textcircled{\alpha}\ f\ t \tag{3.6}$$

Table 3.1: Base equations

Recall from § 2.2.5 that if we fix an A, (TA, μ_A) is a T-algebra that we write as

$$\begin{aligned} \textcircled{\mu} : \forall X.\, (TX \to TA) \to T^2X \to TA \\ \textcircled{\mu}\ X\ f\ t = (\mu \circ T(f))\ t \end{aligned} \tag{3.7}$$

The abstract forms in Table 3.1 have concrete counterparts that are commonplace in the act of manipulating big operators. For example, eq. 3.3 is inspired by equations like

$$\sum_{x \in \{a\}} f(x) = f(a)$$

Equation 3.4 is inspired by equations like

$$\sum_{x \in \{m, \ldots, n\}} f(x+1) = \sum_{x \in \{m+1, \ldots, n+1\}} f(x)$$

This equation is similar to index renaming. We can prove (as in § 2.1) that

$$\begin{aligned}
\sum_{0 \leq k \leq n} (a + bk) &= \sum_{n \leq n-k \leq 0} (a + bk) && \text{(by commutativity)} \\
&= \sum_{0 \leq k \leq n} (a + b(n-k)) && \text{(by 3.4)}
\end{aligned}$$

Equation 3.5 is similar to equations like

$$\max_{y \in \bigcup_{x \in S} g(x)} f(y) = \max_{x \in S} \left(\max_{y \in g(x)} f(y) \right)$$

Equation 3.6 is similar to equations like

$$\sum_{x \in (l_1 ++ \cdots ++ l_n)} f(x) \equiv \sum_{l \in [l_1, \ldots, l_n]} \sum_{x \in l} f(x)$$

We implemented, and thus proved, the base equations in Agda. These equations can be used to reason about big operator expressions derived from arbitrary monads and algebras on $\mathcal{S}etoids$. Deriving these equations in a general form under minimal assumptions allows us to *naturally* structure our library in a way that corresponds to the underlying structure of the big operators themselves.

Below we present a proof of eq. (3.4) in informal mathematics and in Agda. The remaining proofs are in Appendix A.

$$\begin{aligned}
\textcircled{@}\ (\eta_X(a))\ f &= (\alpha \circ T(f))\ (\eta_X(a)) && \text{(by definition)} \\
&= (\alpha \circ T(f) \circ \eta_X)\ a && \text{(by definition)} \\
&= (\alpha \circ \eta_A \circ f)\ a && \text{(by eq. (2.11): naturality of } \eta\text{)} \\
&= (\mathbf{1}_A \circ f)\ a && \text{(by eq. (2.16): unit law of } \alpha\text{)} \\
&= f(a) && \text{(by eq. (2.7))}
\end{aligned}$$

The proof falls out from η being a natural transformation and α a T-algebra. The proof using equational reasoning in Agda follows the same steps, with Setoid.refl A corresponding to "by definition", NT.commute M−η to naturality of η, and T−algebra.unit *T* to the unit law of α. Notice that the mathematical "equality" is now that given by the explicit equivalence relation Setoid._≈_ A, the equality of setoid A.

```
unit : ∀ {X} {x} {f} → Setoid._≈_ A (bigop f ((η X) ⟨$⟩ x)) (f ⟨$⟩ x)
unit {X} {x} {f} =
  begin
    bigop f ((η X) ⟨$⟩ x)
  ≈⟨ Setoid.refl A ⟩
    (α ∘ F₁ f) ⟨$⟩ ((η X) ⟨$⟩ x)
  ≈⟨ Setoid.refl A ⟩
    α ∘ (F₁ f ∘ (η X)) ⟨$⟩ x
  ≈⟨ ∘−resp−≡ʳ {f = F₁ f ∘ (η X)} {h = (η A) ∘ f} {g = α}
    (sym {i = (η A) ∘ f} {j = F₁ f ∘ (η X)} (NT.commute M−η f)) ⟩
    (α ∘ (η A) ∘ f) ⟨$⟩ x
  ≈⟨ ∘−resp−≡ˡ {f = α ∘ (η A)} {h = C.id} {g = f} (T−algebra.unit T) ⟩
    (C.id ∘ f) ⟨$⟩ x
  ≈⟨ C.identityˡ {f = f} ⟩
    f ⟨$⟩ x
  ■
```

3.2 Monads

As mentioned in § 2.1, we use monads to represent container types. Different monads account for different properties of big operators such as associativity, commutativity, and idempotency.

We begin by presenting monads as being "freely" generated from algebraic structures with associativity, commutativity, and idempotency. In §§ 3.2.2 and 3.2.4, we detail the implementation of the list and multiset monads in Agda, as they are arguably the most common container types. In order to represent multisets in Agda, we implemented a permutation library (§ 3.2.3). In §§ 3.2.5 and 3.2.6, we discuss a categorical representation of commutativity and derive some general equations from it.

3.2.1 Free monads

Big operators operate on elements in well-defined collections such as trees, lists, multisets, and sets. Each kind of collection has different properties. For example, binary trees have a fully bracketed structure, lists have an associative structure, multisets "ignore" ordering (commutativity), and sets further ignore repeated elements (idempotency).

In category theory, such collection monads are derived from "adjunctions" between a free functor and an underlying functor between the category of setoids and a category of algebraic structures (see Table 3.2). Intuitively, this means that given a set, we can generate the most general algebraic structure of the kind under consideration and show a correspondence between the various collection monads.

Collection monad	Algebraic structure
Binary tree	Semigroup
Finite list	Monoid
Finite multiset	Commutative monoid
Finite powerset	Idempotent, commutative monoid

Table 3.2: Collection monads and their underlying algebraic structure

For example, given a set S, the free monoid is the set S^* of all finite sequences (lists) of elements of S, made into a monoid using concatenation. The empty list *nil* corresponds to the identity element, and *append(x, y)* corresponds to the binary operation. We proved that *append* is associative in § 2.3.5.

The underlying properties of these collection monads correspond to properties of the binary operator (no property, associativity, commutativity, idempotency) lifted to the big operator. Therefore big operators defined on the list monad have associativity freely built in and big operators defined on the multiset monad have commutativity.

We implemented the tree, list, multiset, and powerset monads in Agda. We will focus our discussion on the two more common containers, lists and multisets.

3.2.2 Lists

To implement the list monad, we need to define its structure (T, η, μ) as in § 2.2.4.

Object map of T

We want T to map a setoid X to a setoid with the carrier List X.

In Agda, List is defined as follows:

```
data List {a} (A : Set a) : Set a where
  []  : List A
  _∷_ : (x : A) (xs : List A) → List A
```

Two lists containing values of a setoid X are equal if they are equal pointwise under the equality given by X. We define a relation on lists:

```
data Rel {a b ℓ} {A : Set a} {B : Set b}
     (_∼_ : REL A B ℓ) : REL (List A) (List B) ℓ where
  [] : Rel _∼_ [] []
  _∷_ : ∀ {x xs y ys} (x∼y : x ∼ y) (xs∼ys : Rel _∼_ xs ys) →
    Rel _∼_ (x ∷ xs) (y ∷ ys)
```

The above expresses that two empty lists are equivalent and two lists $x :: xs$ and $y :: ys$ are equivalent if x is equivalent to y under the given equality and, recursively, xs is equivalent to ys. For a list containing elements from a setoid X, we instantiate the relation _∼_ to $\equiv_X$. We proved that List X together with the relation is a setoid in Agda.

Morphism map of T

The morphism map of T takes functions (preserving equality) from $X \longrightarrow A$ to functions (preserving equality) $\mathit{List}\ X \longrightarrow \mathit{List}\ A$. We defined $T(f)$ to be map f and proved it preserves equality.

Functorality of T

To prove that T is a functor, we show that it preserves identity and composition of arrows (recall § 2.2.2). This corresponds to the following proofs (or programs).

```
  identity : ∀ {c l} {A : Setoid c l} {x : List (Setoid.Carrier A)} →
    Rel (Setoid._≈_ A) (map (Fun.id) x) x

  homomorphism : ∀ {c l} {X Y Z : Setoid c l} {f : X ⟶ Y}
    {g : Y ⟶ Z} {x : List (Setoid.Carrier X)} →
```

```
Rel (Setoid._≈_ Z)
  (map (_⟨$⟩_ (g ∘—Π f)) x)
  ((map (_⟨$⟩_ g) ∘ map (_⟨$⟩_ f)) x)
```

identity says that mapping the identity function is the identity and thus preserves the identity arrow of $\mathcal{S}etoids$. homomorphism says that given arrows $f : X \to Y$ and $g : Y \to Z$, mapping (or "lifting") $g \circ f$ is equivalent to lifting f and lifting g separately and then composing them.

Natural transformations: η and μ

Recall that $\eta_X : X \to \mathit{List}\ X$ and $\mu_X : \mathit{List}\ (\mathit{List}\ X) \to \mathit{List}\ X$. We define η_X to be the function that takes an element $x \in X$ and injects it into a list $[x]$ and μ_X to be the function that takes a list of lists and concatenates them. Again, we provide proofs that the functions preserve equality. For example, for μ_X we prove that if two lists of lists *xss* and *yss* are equivalent (each *xs* $\in$ *xss* is pointwise equivalent to each *ys* $\in$ *yss*), then concat *xss* is equivalent to concat *yss*.

```
cong : ∀ {c l} {X : Setoid c l} {l₁ l₂ : List (List (Setoid.Carrier X))} →
  Rel (Rel (Setoid._≈_ X)) l₁ l₂ →
  Rel (Setoid._≈_ X) (concat l₁) (concat l₂)
```

For naturality (eq. (2.11)) of η and μ, we proved the following commute:

$$\begin{tikzcd}
X \arrow[r, "f"] \arrow[d, "{[\_]}"'] & A \arrow[d, "{[\_]}"] \\
\mathit{List}\ X \arrow[r, "\mathit{map}\ f"'] & \mathit{List}\ A
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathit{List}\ (\mathit{List}\ X) \arrow[r, "\mathit{map}\ (\mathit{map}\ f)"] \arrow[d, "concat"'] & \mathit{List}\ (\mathit{List}\ A) \arrow[d, "concat"] \\
\mathit{List}\ X \arrow[r, "\mathit{map}\ f"'] & \mathit{List}\ A
\end{tikzcd}$$

Recall from § 2.2.3 that natural transformations can represent polymorphic functions. The natural transformations η and μ are indexed sets of functions, with $\eta : \forall X.\ X \to \mathit{List}\ X$ injecting an element of any type into a list $\mu : \forall X.\ \mathit{List}\ (\mathit{List}\ X) \to \mathit{List}\ X$ taking a list of lists of any type and concatenating it. The commutativity of the above diagrams ensures that the polymorphic functions "act the same way" on different types.

Identity and associativity

To complete the definition of the monad, we prove the following in Agda.

$$\begin{tikzcd}
\textit{List } X \arrow[r, "{[\_]}"] \arrow[dr, "id"'] & \textit{List (List } X) \arrow[d, "concat"] & \textit{List } X \arrow[l, "{map\ [\_]}"'] \arrow[dl, "id"] \\
& \textit{List } X &
\end{tikzcd}
\qquad
\begin{tikzcd}
\textit{List (List (List } X)) \arrow[r, "concat"] \arrow[d, "map\ concat"'] & \textit{List (List } X) \arrow[d, "concat"] \\
\textit{List (List } X) \arrow[r, "concat"'] & \textit{List } X
\end{tikzcd}$$

In proving that the above diagrams commute, we prove properties of *concat* and its interaction with *map* and [_]. We can not choose any arbitrary function $\beta : \forall X.\, \textit{List}\ (\textit{List}\ X) \to \textit{List}\ X$, such as the constant function that returns the empty list, for μ. These conditions ensure that our monad "behaves nicely".

Instantiating a general big operator equation for lists

We can instantiate eq. (3.6)

$$\textcircled{@}\ f\ \mu_X(s) \ = \ \underset{t\in s}{\textcircled{@}}\ \textcircled{@}\ f\ t$$

to prove $\sum_{x\in xs++xs'} f(x) = \sum_{x\in xs} f(x) + \sum_{x\in xs'} f(x)$, or more generally:

$$\sum_{x\in(l_1++\cdots++l_n)} f(x) \equiv \sum_{l\in[l_1,\ldots,l_n]} \sum_{x\in l} f(x) \tag{3.8}$$

where μ corresponds to concat (informally written $++$).

3.2.3 A permutation library

Recall that a multiset is a set-like object in which ordering is irrelevant but multiplicity is explicitly significant. Two lists are *multiset equivalent* if and only if there is a bijection between them. This is equivalent to the two lists being permutations of each other.

Implementing multisets in Agda requires implementing permutations in Agda. As Agda's standard library does not contain support for permutations, we wrote a library to facilitate reasoning about permutations and thus equivalence between multisets. This library includes over 30 lemmas and proofs and is a permutation library of interest in its own right, independently of the intended application of the multiset monad.

Defining permutations

Dependent types allow types to be predicated on values, expressing what the program does and what invariants are maintained. We can represent list permutations as a type, which is not possible in non-dependently typed languages like ML [26]. Our implementation is based on work by Brady et al. [8] and Rupert Horlick, a Part III student.

Informally, the empty list is the only permutation of the empty list, and a list $x :: xs$ is a permutation of ys if its tail xs is a permutation of ys with head x removed from it. We need to formalize this in the type system. We proceed as follows.

First, we define the type Any.

```
data Any {a p} {A : Set a}
  (P : A → Set p) : List A → Set (a ⊔ p) where
  here  : ∀ {x xs} (px  : P x)      → Any P (x :: xs)
  there : ∀ {x xs} (pxs : Any P xs) → Any P (x :: xs)
```

The Any type is a dependent type, with Any P xs meaning that at least one element in xs satisfies P. Such an element must either be the head of the list (here) or in the tail of the list (there). A value of this type contains an index into xs and a proof that the predicate holds for the indexed element.

We use Any to define the type $x \in xs$ that is inhabited iff there is a proof that $x \in xs$. A value of type $x \in xs$ is an index into xs and a proof of equivalence to x for the indexed element.

```
module Membership {c ℓ : Level} (S : Setoid c ℓ) where
  _∈_ : Carrier → List Carrier → Set _
  x ∈ xs = Any (_≈_ x) xs
```

For example, here refl is a proof that the head of a list is in the list and, given a proof that $y \in xs$, there $y{\in}xs$ is a proof that $y \in (x :: xs)$ for some x. One can think of there as incrementing the index in the list where the element is found.

```
head−in : (x : Carrier) (xs : List Carrier) → x ∈ (x :: xs)
head−in x xs = here refl

tail−in : ∀ {y} → (x : Carrier) (xs : List Carrier) →
  y ∈ xs → y ∈ (x :: xs)
tail−in x xs y∈xs = there y∈xs
```

We can use a proof of $x \in xs$ to remove the element x from the list xs, by using the proof as an index into the list and removing the element by induction over the index. The first case remove [] () is absurd, as no element can be in the empty list. If the element that satisfies the predicate is the head of the list (here), we return the tail. Else, we return the head concatenated with the result of recursively removing the element from the rest of the list.

```
remove : ∀ {x} (xs : List Carrier) → x ∈ xs → List Carrier
remove [] ()
remove (x1 :: xs) (here x≈x)   = xs
remove (x1 :: xs) (there x∈xs) = x1 :: (remove xs x∈xs)
```

Finally, we define permutations:

```
data Perm : List Carrier → List Carrier → Set (c ⊔ ℓ) where
  nil : Perm [] []
  cons : ∀ {x} {xs : List Carrier} {xs′} → (e : x ∈ xs′) →
    Perm xs (remove xs′ e) → Perm (x :: xs) xs′
```

Perm is a dependent type defined inductively with two constructors, nil and cons. For the base case, two empty lists are permutations with their only permutation given by nil. For the inductive case, two lists $x :: xs$ and xs' are permutations if we can provide a proof e that x is in xs', and also that xs is a permutation of xs' with x removed from it (that is, we can inhabit Perm xs (remove xs' e)).

Thus, an element of type Perm xs ys is a verified (by Agda's type checker) *constructive* proof that xs and ys are permutations.

Permutation is an equivalence relation

We need to prove that Perm is an equivalence relation to use it as the multiset equality. The proof of reflexivity is straight-forward—it is simply the identity permutation. The proofs for symmetry and transitivity are complex and omitted, requiring in-depth reasoning about the construction of permutations (more specifically, inverting and composing permutations).

Generally, the more complex the data type, the more information needs to be provided to the type checker and thus the more involved the proofs. As all proofs written in Agda are *constructive*, providing an element of type Perm xs ys requires providing a complete description of how to construct the bijection or manipulate other permutations to obtain a new permutation. This is non-trivial.

The type signatures for reflexivity, symmetry, and transitivity are below.

```
id-perm : (l : List Carrier) → Perm l l
id-perm [] = nil
id-perm (x ∷ xs) = cons (here (Setoid.refl S )) (id-perm xs)

inv-perm : ∀ {xs ys : List Carrier} → Perm xs ys → Perm ys xs

_∘_ : ∀ {xs ys zs} → Perm ys zs → Perm xs ys → Perm xs zs
```

Examples of lemmas

We implemented over 30 lemmas and proofs to facilitate reasoning about permutations. We provide some examples below.

remove-perm states that if *ys* is a permutation of *zs*, $a \in ys$, $b \in zs$, and $a \equiv b$, then the list given by removing a from *ys* is a permutation of the list given by removing b from *zs*. This is not as straightforward as it may first appear because of the *constructive* nature.

```
remove-perm : ∀ {a b : Carrier} ys zs (e₁ : a ∈ ys) (e₂ : b ∈ zs) →
Perm ys zs → a ≈ b → Perm (remove ys e₁) (remove zs e₂)
```

partition-is-perm allows us to rearrange a list into the concatenation of two lists based on a predicate.

```
partition-is-perm : (p : Carrier → Bool) → (as : List Carrier) →
   Perm as (proj₁ (partition p as) ++ proj₂ (partition p as))
```

We defined partition-is-perm inductively. For the base case where *as* is the empty list, the permutation is nil. Given a proof σ of type partition-is-perm *p as*, we can construct a term of type partition-is-perm *p* (*a* ∷ *as*) for some *a* by adding *a* to the first part of σ if the predicate *p a* holds and to the latter if it does not.

3.2.4 Finite multisets

Now that we have a permutation library, we can implement the finite multiset monad, *Mfin*, by defining (T, η, μ). In Agda, we again define the functor T to take a type X to List X (this implies that the multiset is finite [11]). However, we need to change the underlying equality. This is straightforward as we are working in the category of $\mathcal{S}etoids$—we define two lists to be multiset equivalent whenever they are permutations of each other.

Then using partition—is—perm from § 3.2.3, we can write equations like

$$\sum_{1\leq k\leq n} f(k) \equiv \sum_{\substack{1\leq k\leq n \\ k \text{ even}}} f(k) + \sum_{\substack{1\leq k\leq n \\ k \text{ odd}}} f(k) \tag{3.9}$$

We define the morphism map of T to be *map f* as with the list monad. Proving that this preserves equality requires proving that if *xs* and *ys* are permutations, then the result of mapping f over *xs* is a permutation of the result of mapping f over *ys*, as follows:

```
perm—fmap : ∀ {ℓ} {S T : Setoid ℓ ℓ} {xs} {ys} (f : S ⟶ T) →
  Perm S xs ys →
  Perm T (map (_⟨$⟩_ f) xs) (map (_⟨$⟩_ f) ys)
```

Natural transformations: η and μ

The natural transformations η and μ are defined as with lists as [_] and concat respectively. The difference is in the proofs of naturality and of [_] and concat preserving equality ("congruence").

For example, to prove congruence of concat, it is necessary to prove that if two lists of lists *xss* and *yss* are multiset equivalent, then concat *xss* is multiset equivalent to concat *yss*. Two lists of lists *xss* and *yss* are multiset equivalent if there is a permutation of *xss* such that each list within *xss* is a permutation of its corresponding list in *yss* (i.e. $[[a_1, a_2], [b_1, b_2]]$ is equivalent to $[[b_2, b_1], [a_2, a_1]]$). Defining the bijection in this case requires careful manipulation of the permutations and is omitted due to length (constructing the bijection requires reasoning about two "levels" of permutations: the permutation of the list of lists $[l_1, l_2, \dots, l_n]$ as well as the permutation of each list l_i within the list of lists).

Identity and associativity

To complete the definition of Mfin, we prove the identity and associativity laws below.

$$\begin{tikzcd}
\mathit{Mfin}\ X \arrow[r, "{[\_]}"] \arrow[dr, "id"'] & \mathit{Mfin}\ (\mathit{Mfin}\ X) \arrow[d, "\mathit{concat}"] & \mathit{Mfin}\ X \arrow[l, "\mathit{map}\ {[\_]}"'] \arrow[dl, "id"] \\
 & \mathit{Mfin}\ X &
\end{tikzcd}$$

$$\begin{tikzcd}
\mathit{Mfin}\ (\mathit{Mfin}\ (\mathit{Mfin}\ X)) \arrow[r, "\mathit{concat}"] \arrow[d, "\mathit{map\ concat}"'] & \mathit{Mfin}\ (\mathit{Mfin}\ X) \arrow[d, "\mathit{concat}"] \\
\mathit{Mfin}\ (\mathit{Mfin}\ X) \arrow[r, "\mathit{concat}"'] & \mathit{Mfin}\ X
\end{tikzcd}$$

To prove the above is to do formal proofs of program correctness for the union of multisets.

An equation for multisets

We want to support equations like

$$\sum_x \sum_y f(x, y) = \sum_y \sum_x f(x, y)$$

which hold for commutative operators such as $+$ (and hence for multisets), but not for non-commutative operators (and hence not for lists). To derive this generally, we define *strength* and commutative monads.

3.2.5 Strength

Monads used in functional programming correspond to *strong monads* in category theory. Intuitively in the language of type theory, for a given monad (T, η, μ), a strength is a function[3] that for any types A and B maps pairs $A \times TB$ to $T(A \times B)$, with some properties.

$$\mathit{st} : \forall A, B.\ A \times TB \to T(A \times B)$$

The idea being that *st* provides a mechanism by which parameters can be rippled through the monad structure. For instance, of the list monad,

$$\mathit{st}\ (a, [b_1, \dots, b_n]) = [(a, b_1), \dots, (a, b_n)]$$

[3] Strength is really a natural transformation.

Every endofunctor on $\mathcal{S}etoids$ comes with a canonical strength. We proved this in Agda, by defining strength for arbitrary endofunctors in $\mathcal{S}etoids$. We can define a symmetric function

$$\textit{st'} : \forall A, B.\, TA \times B \to T(A \times B)$$

such that, for the list monad, *st'* $([a_1, \dots, a_n], b) = [(a_1, b), \dots, (a_n, b)]$.

3.2.6 Commutative monads

Commutativity of Mfin

The multiset monad is a commutative monad. This means that the following diagram commutes for all objects A and B.

$$\begin{tikzcd}
TA \times TB \arrow[r, "st_{TA,B}"] \arrow[d, "st'_{A,TB}"'] & T(TA \times B) \arrow[r, "T(st'_{A,B})"] & T^2(A \times B) \arrow[d, "\mu_{A\times B}"] \\
T(A \times TB) \arrow[r, "T(st_{A,B})"] & T^2(A \times B) \arrow[r, "\mu_{A\times B}"] & T(A \times B)
\end{tikzcd}$$

Intuitively, this expresses that given $\{[\{[a_1, a_2, \dots, a_n]\}, \{[b_1, b_2, \dots b_n]\}]\}$, we can pair the elements in either order and end up with an equivalent multiset (this is not true for lists as the ordering of elements matters).

$$\begin{bmatrix} (a_1, b_1) & (a_2, b_1) & \dots & (a_n, b_1) \\ (a_1, b_2) & (a_2, b_2) & \dots & (a_n, b_2) \\ \vdots & \vdots & \ddots & \vdots \\ (a_1, b_n) & (a_2, b_n) & \dots & (a_n, b_n) \end{bmatrix} \equiv \begin{bmatrix} (a_1, b_1) & (a_1, b_2) & \dots & (a_1, b_n) \\ (a_2, b_1) & (a_2, b_2) & \dots & (a_2, b_n) \\ \vdots & \vdots & \ddots & \vdots \\ (a_n, b_1) & (a_n, b_2) & \dots & (a_n, b_n) \end{bmatrix} \tag{3.10}$$

Proving this in Agda requires constructing the bijection, which was done by induction over the “rows” in the first “matrix” and “columns” in the second. We showed that if

$$\begin{bmatrix} (a_1, b_2) & (a_2, b_2) & \dots & (a_n, b_2) \\ \vdots & \vdots & \ddots & \vdots \\ (a_1, b_n) & (a_2, b_n) & \dots & (a_n, b_n) \end{bmatrix} \equiv \begin{bmatrix} (a_1, b_2) & \dots & (a_1, b_n) \\ (a_2, b_2) & \dots & (a_2, b_n) \\ \vdots & \ddots & \vdots \\ (a_n, b_2) & \dots & (a_n, b_n) \end{bmatrix}$$

then we can add $[(a_1, b_1), (a_2, b_1), \dots, (a_n, b_1)]$ as a “row” in the first matrix and as a “column” in the second matrix (by weaving in the elements) to compute a permutation.

Commutativity of big operators

In order to write equations like

$$\sum_x \sum_y f(x,y) = \sum_{(x,y)} f(x,y) = \sum_{(y,x)} f(x,y) = \sum_y \sum_x f(x,y)$$

we need a way of lifting an operator to pairs of containers. We define "parametrised big operators" for a T-algebra (A, α) as follows:

$$\begin{aligned}
&\underset{1}{\textcircled{\alpha}} : \forall X, Y.\, (f : X \times Y \to A) \to TX \times Y \to A \\
&\underset{1}{\textcircled{\alpha}}\; X\; Y\; f\; t \overset{\text{def}}{=} (\alpha \circ T(f) \circ st')\; t \\
&\underset{2}{\textcircled{\alpha}} : \forall X, Y.\, (f : X \times Y \to A) \to X \times TY \to A \\
&\underset{2}{\textcircled{\alpha}}\; X\; Y\; f\; t \overset{\text{def}}{=} (\alpha \circ T(f) \circ st)\; t
\end{aligned}$$

By way of example, when α corresponds to $\sum$,

$$\underset{1}{\textcircled{\alpha}}\; f\; ([a_1, a_2, a_3], b)$$

is equivalent to $f(a_1, b) + f(a_2, b) + f(a_3, b)$ and

$$\underset{2}{\textcircled{\alpha}}\; f\; (a, [b_1, b_2, b_3]])$$

is equivalent to $f(a, b_1) + f(a, b_2) + f(a, b_3)$.

We can recover the above definitions from the non-parametrised big operator (see Appendix A).

Now we can consider

$$\underset{2}{\textcircled{\alpha}}\; (\underset{1}{\textcircled{\alpha}} f) \text{ and } \underset{1}{\textcircled{\alpha}}\; (\underset{2}{\textcircled{\alpha}} f)$$

both of type $TX \times TY \to A$ and determine whether or not they are equal. This is true whenever the monad is commutative (we proved this in Agda and in Appendix A on page 82) and so, for instance, available for the multiset monad.

By way of intuition, when α corresponds to $\sum$,

$$\underset{2}{\textcircled{\alpha}}\; (\underset{1}{\textcircled{\alpha}} f)\; ([a_1, a_2], [b_1, b_2])$$

is equivalent to $f(a_1, b_1) + f(a_1, b_2) + f(a_2, b_1) + f(a_2, b_2)$, and

$$\mathop{@}\limits_{1} (\mathop{@}\limits_{2} f)\ ([a_1, a_2], [b_1, b_2])$$

is equivalent to $f(a_1, b_1) + f(a_2, b_1) + f(a_1, b_2) + f(a_2, b_2)$. This corresponds to the equivalence presented in eq. (3.10).

An example equation in Agda

Suppose we have two lists $(tx, ty) = ([x_1, x_2], [y_1, y_2])$ and a function $g : X \times Y \to Z$. We can combine the two lists as in the previous section with g, to produce

$$tz = [g(x_1, y_1), g(x_2, y_1), g(x_1, y_2), g(x_2, y_2)]$$

We can also combine the two lists in another way to produce

$$tz' = [g(x_1, y_1), g(x_1, y_2), g(x_2, y_1), g(x_2, y_2)]$$

From this, we derive a general big operator equation:

$$\mathop{@}\limits_{z \in tz} f(z) \equiv \mathop{@}\limits_{x \in tx} \mathop{@}\limits_{y \in ty} f(g(x, y)) \tag{3.11}$$

In Agda, a corresponding lemma (using propositional equality for simplicity) is formalized as follows:

```
product−rewrite : ∀ {X Y Z : Set c} {f : Z → A} {g : X × Y → Z}
  {tz : Mfin Z} {tx : Mfin X} {ty : Mfin Y} →
  Perm (setoid Z) tz (combine₁ g (tx , ty)) →
        ⊚ f tz ≈ ⊚ (λ x → ⊚ (λ y → f (g (x , y))) ty) tx
```

Notice the precision of the formalization in properly typing all data but also in assuming that *tz* be a permutation of the application of *g* to the combination of *tx* and *ty*.

We will use this in § 4.1 to express the number-theoretic equation

$$\sum_{d|mn} f(d) \equiv \sum_{e|m} \sum_{e'|n} f(ee')$$

assuming m and n are coprime.

Using the lemma and the commutativity of the multiset monad, we can express

$$\sum_{x \in tx} \sum_{y \in ty} f(x, y) \equiv \sum_{(x,y) \in tz} f(x, y) \equiv \sum_{(x,y) \in tz\text{'}} f(x, y) \equiv \sum_{y \in tz} \sum_{x \in tx} f(x, y)$$

where $tz = [(x_1, y_1), (x_2, y_1), (x_1, y_2), (x_2, y_2)]$ when g is the identity. Note that $\sum_{(x,y) \in tz} f(x, y)$ and $\sum_{(x,y) \in tz\text{'}} f(x, y)$ are equivalent to

$$\underset{1}{@}\ (\underset{2}{@}\ f)\ (tx, ty) \quad \text{and} \quad \underset{2}{@}\ (\underset{1}{@}\ f)\ (tx, ty)$$

3.3 Monad algebras

In our framework, the algebra for a big operator combines the values in a monadic type and is the categorical interpretation of a “lifted operator”. Given an arbitrary monoid, we can “lift” the binary operator to the list monad and derive a unique (up to isomorphism) T-algebra. For example, given the $(\mathbb{N}, +, 0)$ monoid, we can derive an algebra that sums over a list. We detail this construction for lists and multisets and then we derive general equations to reason about distributivity based on algebra morphisms.

3.3.1 List algebras from monoids

Recall that an algebra for a monad (T, η, μ) consists of an object A and a morphism $\alpha : TA \to A$ such that the following diagrams commute:

$$\begin{tikzcd} A \arrow[r, "\eta_A"] \arrow[dr, "\mathbf{1}_A"'] & TA \arrow[d, "\alpha"] \\ & A \end{tikzcd} \qquad \begin{tikzcd} T^2 \arrow[r, "T\alpha"] \arrow[d, "\mu_A"'] & TA \arrow[d, "\alpha"] \\ TA \arrow[r, "\alpha"'] & A \end{tikzcd}$$

For the list monad, η_A injects an element $a \in A$ to the singleton list $[a]$ and μ flattens a list of lists into a list.

$$\mu_A([[a_{11}, \dots, a_{1n_1}], \dots, [a_{m1}, \dots, a_{mn_m}]]) = [a_{11}, \dots, a_{1n_1}, \dots, a_{m1}, \dots, a_{mn_m}]$$

By the laws of T-algebras, we have

$$\alpha([a]) = a$$

and

$$\alpha([a_{11},\dots,a_{1n_1},\dots,a_{m1},\dots,a_{mn_m}]) = \alpha(\alpha([a_{11},\dots,a_{1n_1}]),\dots,\alpha([a_{m1},\dots,a_{mn_m}]))$$

Given a monoid $(M,\cdot,\epsilon)$, we can get a *List*-algebra $\gamma : \mathit{List}\ M \to M$ by defining

$$\gamma[m_1,\dots,m_n] = m_1 \cdot \dots \cdot m_n \tag{3.12}$$

This clearly satisfies the laws. We can even recover the monoid from γ, with $\epsilon = \gamma([\,])$ and $x \cdot y = \gamma([x,y])$. It can be proved that every *List*-algebra is of this form for a unique monoid [2].

Implementation in Agda

We define T-algebras in Agda directly from the mathematical definition:

```
record T−algebra {o ℓ e} {C : Category o ℓ e}
  (M : Monad C) : Set (o ⊔ ℓ ⊔ e) where
  field
    A : Obj
    α : F₀ A ⇒ A

    .{unit}  : C [ C [ α ∘ M−η A ] ≡ id ]
    .{struct} : C [ C [ α ∘ M−μ A ] ≡ C [ α ∘ F₁ α ] ]
```

Given a monoid, we can construct a *List*-algebra by defining

```
  algebra−from−monoid : ∀ {c ℓ} → Monoid c ℓ → T−algebra (monad {c} {ℓ})
```

with the carrier of the T-algebra, A, being the setoid of the monoid, and the structure map of the T-algebra, α, having two components: a *function* _⟨$⟩_ that takes a list containing values of type[4] A and returns a value of type A, and a *proof* of equality cong that states that the results of applying _⟨$⟩_ on two equivalent[5] lists are equivalent.

```
  _⟨$⟩_ = λ xs → foldr _●_ ε xs
```

The function _⟨$⟩_ folds the binary operator over the list. We prove that if $l_1 \equiv l_2$ then the value of folding over l_1 is equivalent to the value of folding over l_2.

[4] This is really the carrier type of A, as A is a setoid.

[5] Equivalence is defined using the equality of A.

```
cong : ∀ {e : Carrier} {l₁ l₂ : List Carrier} →
  Rel _≈_ l₁ l₂ → (foldr _•_ e l₁) ≈ (foldr _•_ e l₂)
```

We prove the unit property using the identity property of ε, with $\alpha([a]) \equiv a{\cdot}\epsilon \equiv a$. The struct property says that flattening a list of lists and then summing over the elements is equivalent to summing over each of the lists and then summing over the resulting values. The proofs are in Appendix A. This allows us to instantiate a big operator from an arbitrary monoid, without needing to directly define a T-algebra.

3.3.2 Multiset algebras from commutative monoids

Similarly, we can construct an algebra γ : *Mfin* $M \rightarrow M$ for the multiset monad from an arbitrary *commutative* monoid $(M, \cdot, \epsilon)$. The algebra is of the same form as previously:

$$\gamma\{\![m_1, \ldots, m_n]\!\} = m_1 \cdot \ldots \cdot m_n$$

Thus when defining

```
algebra−from−cmonoid : ∀ {c} → CommutativeMonoid c c →
  T−algebra (mfin−monad {c})
```

the definitions implemented in algebra−from−monoid can be reused to define A, unit, and struct for the T−algebra. The action of α (given by λ *xs* → foldr _•_ ε *xs*) is as before. The crucial difference is the proof of congruence:

```
cong : ∀ {e : Carrier} {l₁ l₂ : List Carrier} →
  Perm l₁ l₂ → (foldr _•_ e l₁) ≈ foldr _•_ e l₂
```

stating that if l_1 and l_2 are multiset equivalent (equal modulo permutation), then $\alpha(l_1) \equiv \alpha(l_2)$, a property that relies on commutativity of the monoid multiplication. We proved this via the following lemma, which uses commutativity to rearrange the order of applying _•_.

```
remove−comm : ∀ {e : Carrier} {xs : List Carrier} {x} →
  (i : x ∈ xs) → foldr _•_ e xs ≈ x • (foldr _•_ e (remove xs i ))
```

algebra−from−cmonoid lets us easily instantiate a big operator on multisets from an arbitrary commutative monoid (many of which, such as $(\mathbb{N}, +, 0)$, are provided in Agda's standard library) and thereby take advantage of the equations provided in our library.

3.3.3 T-algebra homomorphisms

We want to express equations like

$$c^{\sum_{a\in A} f(a)} \equiv \prod_{a\in A} c^{f(a)} \tag{3.13}$$

In the above equation, exponentiation forms a *monoid homomorphism* (a structure preserving map) between the addition monoid $(\mathbb{N}, +, 0)$ and the multiplication monoid $(\mathbb{N}, *, 1)$. This means that $\forall m, n \in \mathbb{N}.\ \exp(x + y) = \exp(x) * \exp(y)$ and also $\exp(0) = 1$. We will show that a monoid homomorphism between two monoids gives rise to a big operator equation of the form above using the categorical notion of *algebra homomorphisms*.

Definition (Monoid homomorphism)**.** A homomorphism between two monoids $(M, *, \epsilon_M)$ and $(N, \cdot, \epsilon_N)$ is a function $f : M \to N$ such that

– $\forall x, y \in M.\ f(x * y) = f(x) \cdot f(y)$

– $f(\epsilon_M) = \epsilon_N$

By defining $q_c(x) = c^x$ (q_c is a monoid homomorphism), we can represent eq. (3.13) as

$$q_c \left(\underset{a\in A}{\textcircled{α}} f(a) \right) \overset{?}{\equiv} \underset{a\in A}{\textcircled{β}} q_c(f(a)) \tag{3.14}$$

or, equivalently,

$$(q_c \circ (\alpha \circ T(f))\ A \overset{?}{\equiv} (\beta \circ T(q_c \circ f))\ A \tag{3.15}$$

for T being *List*. In fact, these equalities hold for any monad T whenever q_c is an *algebra morphism* from (A, α) to (B, β).

Definition (Algebra morphism)**.** Suppose (A, α) and (B, β) are two T-algebras. An arrow $q : A \to B$ is an algebra morphism if

$$q \circ \alpha = \beta \circ T(q) \tag{3.16}$$

That is, the following diagram commutes.

$$\begin{tikzcd}
TA \arrow[r, "\alpha"] \arrow[d, "Tq"'] & A \arrow[d, "q"] \\
TB \arrow[r, "\beta"] & B
\end{tikzcd}$$

Proof of eq. (3.14).

$$\begin{aligned}
q_c\left(\mathop{\textcircled{\alpha}}_{a\in A} f(a)\right) &= (q_c \circ (\alpha \circ T(f))\ A && \text{(by definition)}\\
&= ((q_c \circ \alpha) \circ T(f))\ A && \text{(by eq. (2.6))}\\
&= ((\beta \circ T(q)) \circ T(f))\ A && \text{(by eq. (3.16))}\\
&= (\beta \circ (T(q_c) \circ T(f)))\ A && \text{(by eq. (2.6))}\\
&= (\beta \circ T(q_c \circ f))\ A && \text{(by eq. (2.9))}\\
&= \mathop{\textcircled{\beta}}_{a\in A} q_c(f(a)) && \text{(by definition)}
\end{aligned}$$

□

We prove that whenever q is a monoid homomorphism from $(A, *, \epsilon_A)$ to $(B, \cdot, \epsilon_B)$, we can construct a $List$-algebra morphism q_c as detailed above from (A, α) to (B, β), where α and β are derived from their respective monoids as in § 3.3.1. That is, we prove $q_c \circ \alpha = \beta \circ \textit{List}\ (q_c)$.

Proof. For the empty list,

$$\begin{aligned}
(q \circ \alpha)\ [\,] &= q(\epsilon_A) \qquad\qquad & (\beta \circ \textit{List}\ (q))\ [\,] &= \beta([\,])\\
&= \epsilon_B & &= \epsilon_B
\end{aligned}$$

where the equalities follow from q preserving identity.

For a list of length n,

$$\begin{aligned}
(q \circ \alpha)[a_1, \dots, a_n] &= q(a_1 * \dots * a_n)\\
&= q(a_1) \cdot \dots \cdot q(a_n)
\end{aligned}$$

and

$$\begin{aligned}
(\beta \circ \textit{List}\ (q))\ [a_1, \dots, a_n] &= \beta([q(a_1), \dots, q(a_n)])\\
&= q(a_1) \cdot \dots \cdot q(a_n)
\end{aligned}$$

□

Thus, given a monoid homomorphism between two monoids, we can derive an algebra morphism between two algebras and thereby derive a general big operator equation. This lets us prove equations like

$$c^{\sum_{a\in A} f(a)} \equiv \prod_{a\in A} c^{f(a)} \quad \text{and} \quad \neg(\bigvee_{a\in A} f(a)) \equiv \bigwedge_{a\in A} \neg f(a)$$

where negation $\neg$ is a monoid homomorphism between $(\mathbb{B}, \vee, 0)$ and $(\mathbb{B}, \wedge, 1)$ in the second equation.

Distributivity and T-algebra endomorphisms

When the two T-algebras of an algebra homomorphism are identical, we have an algebra endomorphism. Suppose (A, α) is a T-algebra. An arrow $q : A \to A$ is an algebra endomorphism if

$$q \circ \alpha = \alpha \circ T(q) \tag{3.17}$$

Consider $T = \textit{List}$, and let the corresponding monoid of (A, α) be $(A, \oplus, \overline{0})$. Then the algebra endomorphism q corresponds to a monoid endomorphism q such that $q(a_1 \oplus a_2) = q(a_1) \oplus q(a_2)$ and $q(\overline{0}) = \overline{0}$.

Such monoid endomorphisms arise commonly from a semiring[6] structure $(S, \oplus, \otimes, \overline{0}, \overline{1})$, where $\otimes$ distributes over $\oplus$:

$$x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z)$$

Then if we take $q_c = \lambda a.\ c \otimes a$, we get a monoid endomorphism and thus an algebra endomorphism. To see this, note that

$$q_c(a_1 \oplus a_2) = c \otimes (a_1 \oplus a_2) = (c \otimes a_1) \oplus (c \otimes a_2)$$

and $q_c(\overline{0}) = c \otimes \overline{0} = \overline{0}$.

For an arbitrary semiring structure, we can derive an algebra endomorphism and thus the following equation (a particular case of eq. (3.14)):

$$q_c\left(\underset{a\in A}{@} f(a)\right) \equiv \underset{a\in A}{@} q_c(f(a))$$

For example, for the semiring $(\mathbb{N}, +, \times, 0, 1)$, we get the following equation for left distributivity $\forall c \in \mathbb{N}$:

[6] A semiring $(S, \oplus, \otimes, \overline{0}, \overline{1})$ is a set S such that $(S, \oplus, \overline{0})$ is a commutative monoid, $(S, \otimes, \overline{1})$ is a monoid, $\otimes$ left and right distributes over $\oplus$, and $\forall s \in S.\ s \otimes \overline{0} = \overline{0} \otimes s = \overline{0}$.

$$c * \sum_{x=0}^{n} f(x) \equiv \sum_{x=0}^{n} (c * f(x))$$

We get right distributivity when q_c is of the form $\lambda a.\ a \otimes c$.

In Agda, we implemented a module that takes a semiring and provides a function to construct an algebra morphism q_c for any c. We used eq. (3.14) to derive right and left distributivity lemmas. When instantiated for propositional equality and multisets, the laws are as follows:

```
l−distrib : ∀ {X} {x : A} {l : Mfin X} {f : X → A} →
   x * ∑ f l ≈ ∑ (λ y → x * f y) l
r−distrib : ∀ {X} {x : A} {l : Mfin X} {f : X → A} →
   (∑ f l) * x ≈ ∑ (λ y → (f y) * x) l
```

These distributivity equations were proved in a more general form in the library, using algebra morphisms as in eq. (3.14). The above laws are instantiations of the general equations in a module that takes a semiring and constructs the corresponding big operator and algebra morphism. Thus, ∑ and * are defined from the particular semiring given, which is not necessarily $(\mathbb{N}, +, \times, 0, 1)$.

3.4 Usability

In the previous sections, we took common algebraic equations using big operators and categorically derived general equations using our framework of monad algebras. We implemented (and thus formally proved) these equations in Agda. However, as a consequence of implementing the equations in their full generality, it can be difficult for practitioners to use the equations in application to reason about simple big operators such as $\sum$, $\prod$, or the Boolean $\bigvee$ and $\bigwedge$. This section gives an overview of steps taken, after evaluating common use cases, to increase the usability of the library.

3.4.1 From the language of algebra to category theory

Familiarity with basic terms of algebra such as monoids, semirings, and homomorphisms is arguably a more reasonable assumption than familiarity with category theory terms for the average Agda user. As such, we defined lemmas that "translate" between the languages of algebra and category theory.

For example, to instantiate a big operator, we need a monad and an algebra. We wrote lemmas to construct an algebra for the list monad from an arbitrary monoid (§ 3.2.2) and an algebra for the multiset monad from an arbitrary commutative monoid (§ 3.2.4). This lets us instantiate a big operator by simply providing a monoid structure, common implementations of which exist in Agda's standard library.

Similarly, rather than having to construct an algebra homomorphism (§ 3.3.3), the user can provide a monoid homomorphism or a semiring structure to use the distributivity equations.

In essence, although category theory is used to develop the underlying theoretical framework and structure the library in order to derive equations in a general way, the library was built to be usable without knowledge of category theory.

3.4.2 Types with propositional equality

This section discusses a technical point regarding the implementation in type theory. For generality, the implementation was based on setoids. Setoids are important in intensional type theories such as that of Agda's [4], where propositional equality is distinguished from definitional equality, as they allow the formation of quotient types (e.g. when constructing the multiset or powerset types or defining the rational numbers).

The cost of this is unnecessary complexity when reasoning about types with propositional equality, such as the natural numbers or Booleans. Our big operators were defined to take *functions that preserve equality* on the setoid. This is a particular type in Agda, so an expression like

$$\sum\ (\lambda\ d \to \sum\ (\lambda\ d' \to f\,(d) * f\,(d')\ )\ (\mathsf{divisors}\ n))\ (\mathsf{divisors}\ m)$$

to express $\sum_{d|m} \sum_{d'|n} f(d) \cdot f(d')$ is not well-typed (as the λ expression is not of the "function that preserves equality" type).

To remedy this, we wrote a module instantiating the big operator equations to work with the Agda type Set instead of Setoid, and with Agda functions instead of "functions that preserve equality".

This changes the type of the big operator from

$$\circledcirc : \forall\ \{X : \mathsf{Setoid}\ c\ \ell\} \to (f : X \longrightarrow \mathsf{A}) \to (t : \mathsf{Setoid.Carrier}\ (\mathsf{F}_0\ X)) \to \mathsf{Setoid.Carrier}\ \mathsf{A}$$

as defined in § 3.1.2, where the *long arrow* indicates that f is of the "function that preserves equality" type, to

$$\circledcirc : \forall\ \{X : \mathsf{Set}\ c\} \to (f : X \to \mathsf{A}) \to (xs : \mathsf{F_0}\ X) \to \mathsf{A}$$

The big operator equations were instantiated similarly. Thus, the above expression for $\sum_{d|m} \sum_{d'|n} f(d) \cdot f(d')$ is now well-typed. This simplifies proof-writing as it removes the need to write the equality proofs associated with setoids when propositional equality is used.

3.4.3 List comprehensions

In evaluating the usability of the library, we considered the ease with which users can create containers. By using lists as the underlying structure of our collection monads, users can take advantage of over 2000 lines of code in the standard library to reason about lists. Lists can be constructed using *interval*, *filtering*, and *partitioning* operations. For example, to define $\sum_{d|m} f(d)$, we might construct the list corresponding to the interval $(1 \leq k \leq m)$ and then filter it with the predicate $\lambda d \to d|?m$.

The standard library provides support for such common list operations, for instance via the following:

$$\mathsf{filter} : \forall\ \{a\}\ \{A : \mathsf{Set}\ a\} \to (A \to \mathsf{Bool}) \to \mathsf{List}\ A \to \mathsf{List}\ A$$

$$\mathsf{partition} : \forall\ \{a\}\ \{A : \mathsf{Set}\ a\} \to (A \to \mathsf{Bool}) \to \mathsf{List}\ A \to (\mathsf{List}\ A \times \mathsf{List}\ A)$$

We discuss filtering and partitioning below.

Filtering. When filtering by a predicate $p : A \to \mathsf{Bool}$, the resulting type is List, which does not contain information (that is, a proof) that the elements in the resulting list satisfy the predicate. For instance,

$$\sum_{\substack{1 \leq k \leq n \\ k \text{ even}}} \lfloor k/2 \rfloor = \sum_{\substack{1 \leq k \leq n \\ k \text{ even}}} k/2$$

holds because of the particular collection of values being even, but this fact can not be used directly without a corresponding proof and big operator equation. The standard library provides a lemma to construct a proof that all elements in a filtered list satisfy the predicate. We provide a lemma all—f

```
all–f : ∀ {X} → (f g : X → A) → (xs : Mfin X) →
  All (λ x → f x ≈ g x) xs → ⊚ f xs ≈ ⊚ g xs
```

which roughly states

$$(\forall x \in t.\ f(x) = g(x)) \Rightarrow \textcircled{@}\ f\ t = \textcircled{@}\ g\ t \tag{3.18}$$

This allows us to reason about equivalence of filtered big operators.

Partitioning. To handle partitioning a collection and express equations like

$$\sum_{1 \leq k \leq n} f(k) \equiv \sum_{1 \leq k \leq n; k \text{ even}} f(k) + \sum_{1 \leq k \leq n; k \text{ odd}} f(k) \tag{3.19}$$

we proved the following:

```
partition–lemma : ∀ {X : Set c} {f : X → A} {t : Mfin X} →
  (p : X → Bool) →
  ⊚ f t ≈ ⊚ f (proj₁ (partition p t)) • ⊚ f (proj₂ (partition p t))
```

The proof of partition–lemma comes from being able to show that partitioning a list on a predicate gives a permutation

```
partition–is–perm : (p : Carrier → Bool) → (as : List Carrier) →
  Perm as (proj₁ (partition p as) ++ proj₂ (partition p as))
```

and the μ-property from eq. (3.6), of which the propositional equality form is below:

```
concat–property : ∀ {X : Set c} {f : X → A} {t : Mfin (Mfin X)} →
  ⊚ f (concat t) ≈ ⊚ (⊚ f) t
```

A particular instance of concat–property is when t contains only two lists, in which case the following holds:

```
concat–lemma : ∀ {X : Set c} {f : X → A} {t₁ t₂ : Mfin X} →
  ⊚ f (t₁ ++ t₂) ≈ (⊚ f t₁) • (⊚ f t₂)
```

Then

```
partition-lemma {X} {f=f} {t} p =
  begin
    ⊚ f t
  ≈⟨ cong₂ refl (partition-is-perm (setoid X) p t) ⟩
    ⊚ f (proj₁ (partition p t) ++ proj₂ (partition p t))
  ≈⟨ concat-lemma {X} {f} {proj₁ (partition p t)} ⟩
    ⊚ f (proj₁ (partition p t)) • ⊚ f (proj₂ (partition p t))
  ∎
```

This is a general lemma for partitioning equations such as eq. (3.19).

3.5 Summary of equations

In this chapter, we summarised the theory and Agda implementation of a general framework for big operators. We derived big operator equations in a unified way, and outlined how the library maintains its usability. Table 3.3 is a generalised version of Table 1.1 on page 6, presenting general forms of big operator equations derived categorically and implemented in the library.

Equation	
Unit identity[1]	$\underset{x\in\eta_X(a)}{\textcircled{@}} f(x) = f(a)$
Composition	$\underset{x\in S}{\textcircled{@}} f(g(x)) = \underset{x'\in T(g)S}{\textcircled{@}} f(x')$
μ-identity	$\underset{x\in\mu_X(S)}{\textcircled{@}} f(x) = \underset{t\in S}{\textcircled{@}}\underset{x\in t}{\textcircled{@}} f(x)$
Distributive law[2]	$g\left(\underset{x\in t}{\textcircled{@}} f(x)\right) = \underset{x\in t}{\textcircled{@}} g(f(x))$
Homomorphism[3]	$h\left(\underset{k\in K}{\textcircled{@}} f(k)\right) = \underset{k\in K}{\textcircled{\beta}} h(f(k))$
Commutative law[4]	$\bigoplus\limits_{x} \bigoplus\limits_{y} f(x,y) = \bigoplus\limits_{y} \bigoplus\limits_{x} f(x,y)$
Partition law	$\bigoplus\limits_{k\in S} f(k) = \bigoplus\limits_{\substack{k\in S\\ p(k)}} f(k) \oplus \bigoplus\limits_{\substack{k\in S\\ \neg p(k)}} f(k)$
Product law[5]	$\bigoplus\limits_{k\in T(g)(\Delta(X\times Y))} f(k) = \bigoplus\limits_{x\in X} \bigoplus\limits_{y\in Y} f(g(x,y))$
Commutativity	$\bigoplus\limits_{k\in K} f(k) = \bigoplus\limits_{k\in \text{perm}(K)} f(k)$
Distributive product[6]	$\bigoplus\limits_{(x,y)\in X\times Y} (f(x)\otimes g(y)) = \left(\bigoplus\limits_{x\in X} f(x)\right) \otimes \left(\bigoplus\limits_{y\in Y} g(y)\right)$
Filtering lemma[7]	$\underset{\substack{k\in S\\ p(k)}}{\textcircled{@}} f(k) = \underset{\substack{k\in S\\ p(k)}}{\textcircled{@}} g(k)$

Table 3.3: Examples of general equations proved in the implementation

[1] $\textcircled{@}$ denotes an arbitrary algebra for an arbitrary monad.
[2] g is an algebra endomorphism on the $\textcircled{@}$ algebra.
[3] h is an algebra homomorphism from the $\textcircled{@}$ algebra to the $\textcircled{\beta}$ algebra.
[4] $\bigoplus$ denotes an algebra for a commutative monad.
[5] Δ denotes the combinator presented in § 3.2.6.
[6] $\oplus$ and $\otimes$ have a semiring structure.
[7] Assuming $\forall k.\ p(k) \Rightarrow f(k) \equiv g(k)$.

Chapter 4

Evaluation

Contents

Chapter 3 described a theoretical framework for big operators and an associated Agda implementation, concluding with a discussion of the implementation's usability. This chapter continues the evaluation of the framework and implementation. To guide our discussion and demonstrate use of the library, we first present a demonstrative proof in § 4.1. We then discuss the practicality and usability of the library in § 4.2 and the expressibility of the framework in § 4.3. Finally, we evaluate the correctness of the theory and implementation in § 4.4.

4.1 A demonstrative proof in Agda

The following is a proposition and proof from number theory, presented in and used throughout Alan Baker's *A concise introduction to the theory of numbers* [3].

Proposition. A function f defined on the positive integers is multiplicative if $f(m)f(n) = f(mn)$ for all m, n with m and n coprime. Suppose f is multiplicative. Let

$$g(n) = \sum_{d|n} f(d)$$

where the sum is over all divisors d of n. Then g is a multiplicative function.

Proof. If m and n are coprime, then

$$g(mn) = \sum_{d|m}\sum_{d|n} f(dd') = \sum_{d|m} f(d) \sum_{d'|n} f(d') = g(m)g(n)$$

□

This one line proof is more involved in formal mathematics, requiring instantiating the big operator with multisets and invoking various divisibility, commutativity, and distributivity lemmas. We demonstrate the step-by-step process of going from a proposition to a proof using our library.

4.1.1 Defining the proposition in Agda

Coprimality. The standard library provides support for coprimes. For $m, n \in \mathbb{N}$, Coprime *m n* is the type that for all $i \in \mathbb{N}$, if $i|m$ and $i|n$, then $i = 1$. Coprime *m n* is inhabited, meaning we can provide a term of the type, if and only if *m* and *n* are coprime.

```
Coprime : (m n : ℕ) → Set
Coprime m n = ∀ {i} → i ∣ m × i ∣ n → i ≡ 1
```

Multiplicative functions. The type of multiplicative functions can be defined straightforwardly from the mathematical definition. Again, isMultiplicative *f* is inhabited iff *f* is multiplicative.

```
isMultiplicative : (f : ℕ → ℕ) → Set
isMultiplicative f = ∀ {m n} → Coprime m n → (f m) * (f n) ≡ f (m * n)
```

Divisors. Given $n \in \mathbb{N}$, we generate the divisors of n by taking the interval $1 \leq k \leq n$ and filtering using the decidable predicate λ *d* → *d* ∣? *n*.

```
divisors : (n : ℕ) → Mfin ℕ
divisors n = filter (⌊_⌋ ∘ (λ d → d ∣? n)) (interval 1 n)
```

We check this behaves as expected by providing a proof refl that divisors 20 ≡ 1 ∷ 2 ∷ 4 ∷ 5 ∷ 10 ∷ 20 ∷ [] (which shows the type is inhabited).

```
pf₁ : divisors 20 ≡ 1 ∷ 2 ∷ 4 ∷ 5 ∷ 10 ∷ 20 ∷ []
pf₁ = refl
```

Instantiating the big operator. As $(\mathbb{N}, +, 0)$ is a commutative monoid, we open the module corresponding to the big operator for multisets, providing the standard library term +-commutativeMonoid to instantiate the big operator to $\sum$. The module provides lemmas parametrised for propositional equality (see § 3.4.2), which is sufficient for natural numbers, and eliminates the need for the user to provide proofs that functions preserve equality.

```
open import Bigop.Lemmas.Mfin
  using (module Lemmas; module SemiringLemmas)
open Lemmas (+-commutativeMonoid) renaming (⊚ to ∑)
open SemiringLemmas (semiring)
```

Defining $g_f(n) = \sum_{d|n} f(d)$. We define $g_f(n)$ as divSum below.

```
divSum : (f : ℕ → ℕ) → (n : ℕ) → ℕ
divSum f n = ∑ f (divisors n)
```

Defining the proposition. We translate the statement of the proposition to Agda.

```
prop : ∀ {m n : ℕ} → Coprime m n →
  (f : ℕ → ℕ) → isMultiplicative f →
  divSum f (m * n) ≡ divSum f m * divSum f n
```

The type is a proposition stating that $\forall m, n \in \mathbb{N}$, if we can provide a proof that m and n are coprime, then for all multiplicative functions f, $g_f(m * n) = g_f(m) * g_f(n)$ and hence g is multiplicative.

4.1.2 Proving the proposition

An informal step-by-step proof

Below is a "pen-and-paper" proof of the proposition where each manipulation of the expression is explicit.

$$
\begin{aligned}
g(mn) &= \sum_{d|mn} f(d) && \text{(by definition)} \quad (4.1a)\\
&= \sum_{d|m} \sum_{d'|n} f(dd') && \text{(by } (*) \text{ below)} \quad (4.1b)\\
&= \sum_{d|m} \sum_{d'|n} f(d)f(d') && (f \text{ multiplicative}) \quad (4.1c)\\
&= \sum_{d|m} \left(f(d) \sum_{d'|n} f(d') \right) && \text{(distributivity with } f(d)) \quad (4.1d)\\
&= \left(\sum_{d|m} f(d) \right) \left(\sum_{d'|n} f(d') \right) && \text{(distributivity with } \sum_{d'|n} f(d')) \quad (4.1e)\\
&= g(m)g(n) && \text{(by definition)} \quad (4.1f)
\end{aligned}
$$

$(*)$: A proof of eq. (4.1b) is based on defining a bijection

$$\{d \in \mathbb{N} \mid d|mn\} \cong \{(d, d') \mid d|m \wedge d'|n\}$$

which holds for m and n coprime.

Formalizing the proof in Agda

Proof in Agda. The informal proof can be formalized in Agda using its equational reasoning. Each of the steps in the informal proof corresponds to an equality in Agda.

```
prop {m} {n} coprime f isMultiplicative =
  begin
    g (m * n)
  ≡⟨ refl ⟩
    ∑ f (divisors (m * n))
  ≡⟨ product-rewrite {tx = divisors m} lemma₂ ⟩
    ∑ (λ d → ∑ (λ d′ → f (d * d′) ) (divisors n)) (divisors m)
  ≡⟨ sym (all-f (λ d → ∑ (λ d′ → f d * f d′) (divisors n))
    (λ d → ∑ (λ d′ → f (d * d′)) (divisors n))
    (divisors m) lemma) ⟩
    ∑ (λ d → ∑ (λ d′ → f (d) * f (d′) ) (divisors n)) (divisors m)
  ≡⟨ cong₂ {t₁ = divisors m}
    (λ {x} → sym (l-distrib {x = f x} {l = divisors n}))
    Mfin-refl ⟩
    ∑ (λ d → (f d) * ∑ f (divisors n)) (divisors m)
  ≡⟨ sym (r-distrib {x = ∑ f (divisors n)} {l = divisors m}) ⟩
    (∑ f (divisors m)) * (∑ f (divisors n))
  ≡⟨ refl ⟩
    g(m) * g(n)
  ∎
  where
  g = divSum f
```

The first step, $\mathsf{g}\ (m * n) \equiv \sum f\ (\mathsf{divisors}\ (m * n))$, follows from the definition of g so the proof is given by propositional equality, refl.

The next step corresponds to eq. (4.1b). The idea is that we want to express the multiset divisors $(m * n)$ as two multisets, divisors m and divisors n, that can be combined with

```
combine₁ : ∀ {X Y Z} (g : X × Y → Z) (tx×ty : Mfin X × Mfin Y) → Mfin Z
combine₁ g tx×ty = (map g ∘ concat ∘ map st₁ ∘ st₂) tx×ty
```

where g in this case corresponds to $\lambda(x, y).\ x * y$.

For example, $\mathsf{combine}_1$ on $([a_1, a_2], [b_1, b_2])$ would return $[g(a_1, b_1), g(a_1, b_2), g(a_2, b_1), (a_2, b_2)]$ (the lists can also be combined the other way, which was proved equivalent in § 3.2.6).

Assuming we have a proof of the divisors of mn being a permutation of the divisors of m and n combined using multiplication with $\mathsf{combine}_1$ (e.g. for $m = 2$ and

$n = 3$, this is a proof that $[1, 2, 3, 6]$ is a permutation of $[1 * 1, 1 * 3, 2 * 1, 2 * 3]$)

```
lemma₂ : Perm (setoid ℕ) (divisors (m * n))
  (combine₁ (uncurry _*_) (divisors m , divisors n))
```

then product−rewrite (from page 46)

```
product−rewrite : ∀ {X Y Z : Set c} {f : Z → A} {g : X × Y → Z}
  {tz : Mfin Z} {tx : Mfin X} {ty : Mfin Y} →
  Perm (setoid Z) tz (combine₁ g (tx , ty)) →
  ⊚ f tz ≈ ⊚ (λ x → ⊚ (λ y → f (g (x , y))) ty) tx
```

can be used to prove eq. (4.1b).

To prove $\sum_{d|m} \sum_{d'|n} f(dd') = \sum_{d|m} \sum_{d'|n} f(d)f(d')$, we use the "filtering lemma" from Table 3.3 and eq. (3.18) with the following:

```
lemma : All (λ x → (∑ (λ d′ → f x * f d′) (divisors n) ≡
  ∑ (λ d′ → f (x * d′)) (divisors n))) (divisors m)
```

which states that $\forall d \in$ divisors m, $\sum_{d'|n} f(dd') = \sum_{d'|n} f(d)f(d')$. The proof is omitted here. The idea is that we carry along the proofs that $\forall d \in$ divisors m, $d|m$ and $\forall d' \in$ divisors n, $d'|n$ such that the multiplicative function property of f can be invoked.

The remaining equalities are straightforward using the equations provided by the library, invoking distributivity of multiplication over addition.

4.2 Practicality and usability of the library

We demonstrated the process of going from a mathematical proposition to a formal proof in Agda using our library. Defining the proposition in Agda (implementing divisors and instantiating the big operator) was straightforward, with the big operator definitions and equations available from importing a module parametrised by a monoid defined in the standard library.

The proof demonstrated the three ideas regarding usability outlined in § 3.4:

1. Replacing the language of category theory with that of algebra: the proof required no knowledge of category theory and used only ideas from algebra such as distributivity.

2. Providing an interface for types with propositional equality: there was no mention of setoids or "functions that preserve equality" in the proof. This allows the user to use λ functions and simplifies proof-writing.

3. Providing lemmas to reason about list comprehensions: the proof demonstrated the interaction of filtering (used in divisors) with big operator equations that are true based on a property of the values in the container (used with all−f).

As desired, the steps in the Agda proof correspond directly to the steps in a pen-and-paper proof that separated each manipulation of the expression into a separate equation. This demonstrates the expressibility of the library, which provides a rich set of equations to manipulate big operator expressions.

A challenge in terms of usability is due to the use of permutations to define multiset equality. Constructing a permutation each time commutativity is invoked may not be straightforward. However, often big operator expressions are not arbitrarily rearranged and are instead rearranged based on some predicate. We provide support for many such manipulations with our permutation library.

4.3 A discussion of equations

Table 3.3 on page 58 (a general version of Table 1.1 on page 6) illustrates common big operator equations proved in the implementation.

Equations expressed with @ hold regardless of the monad or algebra used. These equations were proved once only in the language of category theory and can be invoked without needing to reprove the equation for a specific monad or algebra.

Equations based on commutativity (expressed with $\bigoplus$) were derived using the notion of a *commutative monad*. Thus, these equations can be used for any commutative monad (such as the multiset or powerset).

The distributive law and homomorphism law were also proved abstractly using the idea of algebra endomorphisms and homomorphisms respectively. We proved that the equations arose out of semiring structures and monoid homomorphisms. In deriving these equations, we developed theory on the representation of algebraic expressions, structures, and morphisms in category theory.

The development of the library was motivated by examining common uses of big operator expressions. We demonstrated the process of taking an algebraic expression and deriving a general equation in our framework. Overall, these equations cover the vast majority of the equations presented in chapter two of Knuth's *Concrete Mathematics* [13].

4.4 Correctness of the theory and implementation

Type theories are a class of formal systems. The implementation of our categorical framework in Agda's type theory serves to formally verify its correctness and ensure the careful handling of issues often ignored in informal mathematics. The informal categorical proofs and equations presented throughout Chapter 3 were formally proved correct by implementing type-checked proofs in Agda.

Instantiating the list and multiset monads in §§ 3.2.2 and 3.2.4 required formally proving various monad laws related to list concatenation and multiset union. Similarly, constructing the algebras required proving the T-algebra laws. This is in contrast to languages like ML and Haskell, where the monads and algebras can be implemented but not proved correct (that is, it can not be proved that the monad laws are obeyed and this is left at the discretion of the programmer).

Finally, throughout the development of the project, unit testing was used to increase confidence that the library behaved as expected. Unit testing was achieved using Agda's type system, where tests were represented as proofs of propositional equality. The equality represented the expression and the "desired" result, with the proof simply being refl. The type checker would raise an error if the test failed.

4.5 Success criteria

I have met all my success criteria and implemented two extensions.

> *Developed and formalized theory for a general framework for big operators on algebras of monads.*

Chapter 3 presented big operators in terms of monad algebras and developed in the framework the theory for operator properties like associativity, distributivity, and commutativity. This has not been done before at the abstract level of category theory. I derived general equations to reason about algebraic structures such as monoids and semirings.

> *Designed and implemented a generic framework for big operators in Agda based on the theory of algebras of monads, having the abil-*

> *ity to represent and define big operators with different properties and monads of different types such as lists or sets.*

I developed a library, consisting of over 30 modules and over 2500 lines of Agda "code" structured using the categorical framework. Table 3.3 provides examples of general equations derived and implemented in the library. I implemented four different types of monads: the tree, list, multiset, and powerset monads.

> *Demonstrated the use of the library by proving various equivalences using big operators such as results on summations from* Concrete Mathematics *[13].*

I implemented nearly all of the equations in chapter two (on Σ notation) of *Concrete Mathematics* and many others, as summarised in Table 3.3. I demonstrated use of the library by writing proofs, such as that for number-theoretic multiplicative functions in § 4.1.

> *Instantiate and prove results using non-standard monads such as multisets, indexed containers, or binary trees.*

I proved lemmas for four monads including the multiset and binary tree. Identifying the correspondence between container types and algebraic structures, and thus the correspondence between the multiset and commutative monoid, was part of the theoretical work of this project. This made multisets an essential part of the implementation in Agda and resulted as a by-product in the creation of a library to reason about permutations (which can not be implemented as a type in non-dependently typed languages like ML).

> *Formally verify the correctness of the implementation.*

When instantiating the monads and algebras, I proved that the monad and algebra laws hold. This formally proved the correctness of the implementation. This is in contrast to Haskell and ML, where monads and algebras can be defined but not formally proved correct. In addition, the implementation in Agda's type system of the theory developed in this dissertation serves as formal certification.

4.6 Summary

This chapter demonstrated that the project has exceeded its success criteria and that the proposed framework can be used to structure an implementation of big operators in Agda that is practical while deriving equations in a general way.

Chapter 5

Conclusion

Contents

5.1 Results

This dissertation developed theory for a more general approach to big operators than has been taken previously (c.f. [14, 15, 6]).

This project addressed each of the challenges presented in Chapter 1.

> **Problem I. Formalization of big operators**
> We developed and formalized a general framework for big operators in terms of monads and their algebras, using category theory to provide a generalized notion of functions and a rich conceptual background for reasoning about monads and their algebras.
>
> **Problem II. Manipulation of big operator expressions**
> From specific big operator equations as in Table 1.1 on page 6, we derived a rich set of generalized equations (presented in Table 3.3 on page 58) in a unified way. We developed theory for operator properties such as associativity, distributivity, and commutativity in our categorical framework.

Problem III. Formal implementation
We implemented our framework in Agda's type theory for formal verification. We implemented and instantiated a variety of monads and operator properties, and also developed an independent permutation library. We present a big operator library designed and implemented in a modular way, with lemmas and big operator equations for commonly used algebraic structures such as monoids and semirings.

Problem IV. Usability
We demonstrated how the theoretical framework could be used to develop a big operator library that is both usable and well-structured in a dependently-typed language. Despite big operator notation being ubiquitous, reasoning about big operators is not fully supported in many proof assistants. This project explored a general framework upon which to build a usable big operator library and produced a working prototype in Agda.

5.2 Overview

The formalization of mathematics is challenging for two reasons: firstly, formal proofs require more detail and secondly, the formalized body of mathematics is limited and thus many lemmas need to be formally proved before significant work can be done.

This work developed infrastructure in a modular way in the form of a big operator (and permutation) library. Big operator notation is common in the mathematical vernacular and this library increases the classes of formal proofs that can be naturally expressed.

Moreover, this work demonstrated the process of formalizing an algebraic concept through representing it in different languages: (i) the language of *abstract algebra* when reasoning about familiar big operators, (ii) the language of *category theory* when generalizing big operators and presenting a categorical framework, (iii) the language of *dependent type theory* when providing a more familiar syntax for the former, and (iv) the language of *Agda* when writing type-checked proofs and thereby formalizing big operators.

5.3 Lessons learned

This project was research-like in nature, with the starting point being an abstract definition of big operators in terms of monad algebras [10]. Expressing new equations in the framework involved expressing the algebraic equations using ideas from category theory, and then further formalizing the mathematics by expressing the equations in Agda. Without having prior experience with category theory, proof assistants, Agda, or dependently-typed programming, understanding how to represent and prove big operator equations—first in the language of category theory and then in that of dependent type theory and Agda—was a challenge.

Overall, I have gained an exciting glimpse into the world of research and the beautiful correspondences between mathematical logic, type theory, and category theory.

Bibliography

[1] Andreas Abel. Agda: equality, 2012.

[2] Steve Awodey. *Category theory*. Oxford University Press, 2010.

[3] Alan Baker. *A concise introduction to the theory of numbers*. Cambridge University Press, 1984.

[4] Gilles Barthe, Venanzio Capretta, and Olivier Pons. Setoids in type theory. *J. Funct. Program.*, 13(2):261–293, March 2003.

[5] John L Bell. Types, sets, and categories.

[6] Yves Bertot, Georges Gonthier, Sidi Ould Biha, and Ioana Pasca. *Canonical Big Operators*, pages 86–101. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[7] Andreas Blass. The interaction between category theory and set theory. 1984.

[8] Edwin Brady, Christoph A Herrmann, and Kevin Hammond. Lightweight invariants with full dependent types. *Trends in Functional Programming*, 2007, 2008.

[9] Nils Anders Danielsson. Bag equivalence via a proof-relevant membership relation. In Lennart Beringer and Amy P. Felty, editors, *ITP*, volume 7406 of *Lecture Notes in Computer Science*, pages 149–165. Springer, 2012.

[10] Marcelo Fiore and Timothy G. Griffin. On the algebras of big operators. Unpublished manuscript.

[11] Denis Firsov and Tarmo Uustalu. Dependently typed programming with finite sets. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*, WGP 2015, pages 33–44, New York, NY, USA, 2015. ACM.

[12] Herman Geuvers, Randy Pollack, Freek Wiedijk, and Jan Zwanenburg. The algebraic hierarchy of the fta project.

[13] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1994.

[14] Daniel Gustafsson and Nicolas Pouillard. Counting on type isomorphisms.

[15] Daniel Gustafsson and Nicolas Pouillard. Foldable containers and dependent types.

[16] John Harrison. Hol light: A tutorial introduction. In *International Conference on Formal Methods in Computer-Aided Design*, pages 265–269. Springer, 1996.

[17] John Harrison. HOL Light: An overview. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2009*, volume 5674 of *Lecture Notes in Computer Science*, pages 60–66, Munich, Germany, 2009. Springer-Verlag.

[18] Martin Hofmann. Extensional concepts in intensional type theory. 1995.

[19] Martin Hofmann. *On the interpretation of type theory in locally cartesian closed categories*, pages 427–441. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995.

[20] Max Kelly. *Basic concepts of enriched category theory*, volume 64. CUP Archive, 1982.

[21] Anders Kock. Commutative monads as a theory of distributions. *Theory and Applications of Categories*, 26(4):97–131, 2012.

[22] Joachim Lambek and Philip J Scott. *Introduction to higher-order categorical logic*, volume 7. Cambridge University Press, 1988.

[23] Leonhard Mackert. Big operators in agda. 2015.

[24] P. Martin-Löf. Constructive mathematics and computer programming. In *Proc. Of a Discussion Meeting of the Royal Society of London on Mathematical Logic and Programming Languages*, pages 167–184, Upper Saddle River, NJ, USA, 1985. Prentice-Hall, Inc.

[25] Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory*, volume 9. Bibliopolis Napoli, 1984.

[26] Robin Milner. *The definition of standard ML: revised*. 1997.

[27] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A*

Proof Assistant for Higher-order Logic. Springer-Verlag, Berlin, Heidelberg, 2002.

[28] Ulf Norell. Dependently typed programming in agda. In *Proceedings of the 6th International Conference on Advanced Functional Programming*, AFP'08, pages 230–266, Berlin, Heidelberg, 2009. Springer-Verlag.

[29] Bryan O'Sullivan, John Goerzen, and Donald Bruce Stewart. *Real world haskell: Code you can believe in*. " O'Reilly Media, Inc.", 2008.

[30] Changhee Park, Guy L Steele Jr, and Jean-Baptiste Tristan. Parallel programming with big operators. In *ACM SIGPLAN Notices*, volume 48, pages 293–294. ACM, 2013.

[31] Dan Peebles, James Deikun, Andrea Vezzosi, James Cook, and other contributors. categories. `https://github.com/copumpkin/categories`, 2016.

[32] Andrew M. Pitts. Lecture notes on types for part ii of the computer science tripos. 2016.

[33] John Power and Yoshiki Kinoshita. Category theoretic structure of setoids. 2013.

[34] Dag Prawitz. *Intuitionistic Logic: A Philosophical Challenge*, pages 1–10. Springer Netherlands, Dordrecht, 1980.

[35] Anirudh Sankar. Monads and algebraic structures. Technical report, Technical report, The University of Chicago, Chicago, IL, USA, 2012. URL: http://math. uchicago. edu/˜ may/REU2012/REUPapers/Sankar. pdf, 2012.

[36] Dana S Scott and Christopher Strachey. *Toward a mathematical semantics for computer languages*, volume 1.

[37] Robert AG Seely. Locally cartesian closed categories and type theory. In *Mathematical proceedings of the Cambridge philosophical society*, volume 95, pages 33–48. Cambridge Univ Press, 1984.

[38] Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard isomorphism*, volume 149. Elsevier, 2006.

[39] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.0*, April 2004. `http://coq.inria.fr`.

[40] Paul van der Walt and Wouter Swierstra. Engineering Proof by Reflection in Agda. In *Revised Selected Papers of the 24th International Symposium on Implementation and Application of Functional Languages*, volume 8241 of

Lecture Notes in Computer Science, pages 157–173. Springer International Publishing, 2012.

[41] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 61–78. ACM, 1990.

[42] Philip Wadler. Propositions as types. *Communications of the ACM*, 58(12):75–84, 2015.

Appendix A

Category theory supplementals

Cartesian-closed categories

This section contains supplementary material regarding the correspondence between type theory (or rather, typed lambda calculus) and a particular kind of category called a *Cartesian-closed category*. This is discussed in detail in [19, 37].

If there is a correspondence between type theory and category theory, then we should be able to define familiar type-theoretic constructions such as products, sums, and data structures like lists. Category theory does so by means of abstract universal properties that characterize the structures up to canonical isomorphism. Briefly, we show how the type theoretic notion of products can be represented in category theory.

Definition (Product)**.** A product of two objects A and B in a category $\mathcal{C}$ is an object $A \times B$ with two morphisms[1]

$$(\pi_1)_{A,B} : A \times B \to A \quad (\pi_2)_{A,B} : A \times B \to B$$

such that for any two morphisms $f : X \to A$ and $g : X \to B$ there exists a unique morphism $\langle f, g\rangle : X \to A \times B$ such that $(\pi_1)_{A,B} \circ \langle f, g\rangle = f$ and $(\pi_2)_{A,B} \circ \langle f, g\rangle = g$.

For example, consider the category $\mathcal{P}oset$ where the objects are partially ordered sets of the form $\mathbf{A} = (A, \sqsubseteq_A)$ and the morphisms are monotonic functions $f : \mathbf{A} \to \mathbf{B}$ such that $x \sqsubseteq_A y \Rightarrow f(x) \sqsubseteq_B f(y)$. Posets arise everywhere in computer science when modelling partially defined elements. The product $\mathbf{A} \times \mathbf{B}$ has

[1] We use “arrows” and “morphisms” synonymously.

the underlying set $A \times B$, the Cartesian products of sets A and B, with the partial order

$$(x, y) \sqsubseteq_{A \times B} (x', y') \iff x \sqsubseteq_A x' \land y \sqsubseteq_B y'$$

We can generalize the idea from binary products to arbitrary products. The "terminal object"—the product of an empty family of objects—is denoted **1**.

Definition (Terminal object)**.** The terminal object in a category $\mathcal{C}$ is an object **1** such that there is a unique arrow $!_X : \mathbf{X} \to \mathbf{1}$ from any object **X** of $\mathcal{C}$.

In the category $\mathcal{S}et$, the terminal object is any one-element set. In functional programming and type theory, the terminal object is the *unit type*, denoted `()` in Haskell and Rust, and `unit` in ML.

In the category $\mathcal{S}et$, the collection of functions from A to B forms a set $[A \to B]$, which is an object in the category. Similarly in the category $\mathcal{P}oset$, the collection of monotone functions from **A** to **B** written $[\mathbf{A} \to \mathbf{B}]$ with partial order

$$f \sqsubseteq_{A \to B} g \iff \forall x \in A.\, f(x) \sqsubseteq_B g(x)$$

gives another object $([\mathbf{A} \to \mathbf{B}], \sqsubseteq_{A \to B})$. These function spaces correspond to the idea of *exponentials* in category theory.

Definition (Exponential)**.** For objects A and B in category $\mathcal{C}$, the exponential is an object $A \Rightarrow B$ (or B^A) together with a morphism $\mathit{eval}_{A,B} : (A \Rightarrow B) \times A \to B$ such that for any morphism $f : Z \times A \to B$, there exists a unique morphism $h : Z \to (A \Rightarrow B)$ such that $\mathit{eval}_{A,B} \circ (h \times \mathrm{id}_A) = f$. We denote this unique morphism h familiarly by *curry* f.

Definition (Cartesian-closed category)**.** A category that has all finite products and all exponentials is called a *Cartesian-closed category*, meaning we can form finite products from all objects and for all pairs of objects there is an exponential.

Typed lambda calculus, and thus simple type theory, can be given semantics in a Cartesian-closed category [19].

Monad equivalence

This section provides a proof that the Kleisli triple, which is the more commonly used definition of a monad in functional programming, is equivalent to the monad defined in § 2.2.4.

Definition (Kleisli triple)**.** A *Kleisli triple* $(T, \eta, (-)^*)$ on a category $\mathcal{C}$ consists of

- an operator $T : \mathcal{C} \to \mathcal{C}$
- for each $A \in \mathcal{C}$, an arrow $\eta_A : A \to TA$, and
- for all $A, B \in \mathcal{C}$, an operator $(-)^* : \mathcal{C}(A, TB) \to \mathcal{C}(TA, TB)$

satisfying the following identities:

$$f^* \circ \eta_A = f \tag{A.1}$$

$$\eta_A^* = \mathbf{1}_{TA} \tag{A.2}$$

$$(g^* \circ f)^* = g^* \circ f^* \tag{A.3}$$

Proposition. Monads and Kleisli triples are "equivalent".

Proof. Let (T, η, μ) be a monad on a category $\mathcal{C}$. T and η correspond trivially to the T and η in the Kleisli triple. For $f : A \to TB$ in $\mathcal{C}$, we define f^* to be $TA \xrightarrow{Tf} T^2B \xrightarrow{\mu_B} TB$, or $\mu_B \circ Tf$.

We show the three equations hold. Let $f : A \to TB$. For (A.1):

$$\begin{aligned} f^* \circ \eta_A &= \mu_B \circ Tf \circ \eta_A \\ &= \mu_B \circ \eta_{TB} \circ f && \text{(by naturality)} \\ &= \mathbf{1}_{TB} \circ f && \text{(by 2.14)} \\ &= f \end{aligned}$$

For (A.2):

$$\eta_A^* = \mu_A \circ T\eta_A = \mathbf{1}_{TA} \qquad \text{(by 2.14)}$$

Finally for (A.3), let $f : A \to TB$ as previously and let $g : B \to TD$. Then

$$\begin{aligned} g^* \circ f^* &= (\mu_D \circ Tg) \circ (\mu_B \circ Tf) \\ &= \mu_D \circ \mu_{TD} \circ T^2 g \circ Tf && \text{(by naturality of } \mu \text{ wrt } g\text{)} \\ &= \mu_D \circ T\mu_D \circ T^2 g \circ Tf && \text{(by 2.15)} \\ &= \mu_D \circ T(\mu_C \circ Tg \circ f) \\ &= (g^* \circ f)^* \end{aligned}$$

So $(T, \eta, (-)^*)$ is a Kleisli triple derived from monad (T, η, μ). We can recover the original monad from the Kleisli triple as follows (T and η are trivial):

$$\begin{aligned} \mathbf{1}_{TA}^* &= \mu_{TA} \circ T\mathbf{1}_{TA} \\ &= \mu_{TA} \circ \mathbf{1}_{T^2A} \\ &= \mu_{TA} \end{aligned}$$

Let $(T, \eta, (-)^*)$ be a Kleisli triple on a category $\mathcal{C}$. We want T to be an endofunctor in $\mathcal{C}$. For $f : A \to B$ in $\mathcal{C}$ let $Tf : TA \to TB$ be defined as

$$Tf = (\eta_B \circ f)^*$$

This is a functor as for all objects $A \in \mathcal{C}$

$$T\mathbf{1}_A = (\eta_A \circ \mathbf{1}_A)^* = \eta_A^* = \mathbf{1}_{TA} \qquad \text{(by A.2)}$$

and for arrows $f : A \to B$ and $g : B \to C$ in $\mathcal{C}$,

$$\begin{aligned} Tg \circ Tf &= (\eta_c \circ g)^* \circ (\eta_B \circ f)^* \\ &= ((\eta_c \circ g)^* \circ \eta_B \circ f)^* && \text{(by A.3)} \\ &= (\eta_c \circ g \circ f)^* && \text{(by A.1)} \\ &= T(g \circ f) \end{aligned}$$

To prove η is a natural transformation $\mathrm{id}_{\mathcal{C}} \to T$, we need to prove that for all $f : A \to B$ in $\mathcal{C}$ the following square commutes.

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "\eta_A"'] & B \arrow[d, "\eta_B"] \\
T(A) \arrow[r, "T(f)"'] & T(B)
\end{tikzcd}$$

$$Tf \circ \eta_A = (\eta_B \circ f)^* \circ \eta_A = \eta_B \circ f \qquad \text{(by A.1)}$$

Finally we define $\mu : T^2 \to T$ as

$$\mu_A = id_{TA}^*$$

This is natural since for all $f : A \to B$

$$\begin{aligned} \mu_B \circ T^2 f &= id_{TB}^* \circ (\eta_{TB} \circ Tf)^* \\ &= (id_{TB}^* \circ \eta_{TB} \circ Tf)^* && \text{(by A.3)} \\ &= (id_{TB} \circ Tf)^* && \text{(by A.1)} \\ &= (Tf)^* \\ &= (\eta_B \circ f)^{**} && \text{(by definition of Tf)} \\ &= ((\eta_B \circ f)^* \circ \mathbf{1}_{TA})^* \\ &= (\eta_B \circ f)^* \circ id_{TA}^* && \text{(by A.3)} \\ &= Tf \circ \mu_A \end{aligned}$$

Now we need to show the three monad laws. For the identity laws:

$$\mu_A \circ \eta_{TA} = id^*_{TA} \circ \eta_{TA} = \mathbf{1}_{TA} \qquad \text{(by A.1)}$$

and

$$\begin{aligned}
\mu_A \circ T\eta_A &= \mathbf{1}^*_{TA} \circ (\eta_{TA} \circ \eta_A)^* \\
&= (\mathbf{1}^*_{TA} \circ \eta_{TA} \circ \eta_A)^* && \text{(by A.3)} \\
&= (\mathbf{1}_{TA} \circ \eta_A)^* && \text{(by A.1)} \\
&= \eta^*_A \\
&= \mathbf{1}_{TA} && \text{(by A.2)}
\end{aligned}$$

Finally for (2.15):

$$\begin{aligned}
\mu_A \circ T\mu_A &= \mathbf{1}^*_{TA} \circ (\eta_{TA} \circ \mathbf{1}^*_{TA})^* \\
&= (\mathbf{1}^*_{TA} \circ \eta_{TA} \circ \mathbf{1}^*_{TA})^* && \text{(by A.3)} \\
&= (\mathbf{1}_{TA} \circ \mathbf{1}^*_{TA})^* && \text{(by A.1)} \\
&= \mathbf{1}^{**}_{TA} \\
&= (\mathbf{1}^*_{TA} \circ \mathbf{1}_{T^2A})^* \\
&= \mathbf{1}^*_{TA} \circ \mathbf{1}^*_{T^2A} && \text{(by A.3)} \\
&= \mu_A \circ \mu_{TA}
\end{aligned}$$

So (T, η, μ) is a monad defined from Kleisli triple $(T, \eta, (-)^*)$. We can recover the original Kleisli triple. T and η are trivial. For $(-)^*$:

$$\begin{aligned}
\mu_B \circ Tf &= \mathbf{1}^*_{TA} \circ (\eta_B \circ f)^* \\
&= (\mathbf{1}^*_{TA} \circ \eta_B \circ f)^* && \text{(by A.3)} \\
&= (\mathbf{1}_{TA} \circ f)^* && \text{(by A.1)} \\
&= f^*
\end{aligned}$$

So the Kleisli triple and monad are equivalent definitions. □

Proofs of general big operator equations

This section provides category theory proofs of the base equations in Table 3.1.

$$\begin{aligned}
\textcircled{@}\ \mathrm{id}_A\, t &= (\alpha \circ T(\mathrm{id}_A))\, t \\
&= \alpha(t)
\end{aligned}$$

$$\begin{aligned}
\textcircled{\alpha}\ f\ (\eta_X(a)) &= (\alpha \circ T(f))\ (\eta_X(a)) \\
&= (\alpha \circ T(f) \circ \eta_X)\ a \\
&= (\alpha \circ \eta_A \circ f)\ a \\
&= (\mathbf{1} \circ f) \\
&= f(a)
\end{aligned}$$

$$\begin{aligned}
\textcircled{\alpha}\ (f \circ g)\ t &= (\alpha \circ T(f \circ g))\ t \\
&= (\alpha \circ (T(f) \circ T(g)))\ t \\
&= (\alpha \circ T(f))\ (T(g)(t)) \\
&= \textcircled{\alpha}\ f\ (T(g)(t))
\end{aligned}$$

$$\begin{aligned}
\textcircled{\alpha}\ f\ (\textcircled{\mu}\ g\ t) &= (\alpha \circ T(f))(\mu \circ T(g))\ t \\
&= (\alpha \circ T(f) \circ \mu \circ T(g))\ t \\
&= (\alpha \circ \mu \circ T^2(f) \circ T(g))\ t \\
&= (\alpha \circ T\alpha \circ T^2(f) \circ T(g))\ t \\
&= (\alpha \circ T(\alpha \circ T(f) \circ g))\ t \\
&= \textcircled{\alpha}\ (\lambda x. \textcircled{\alpha}\ g(x)\ f)\ t
\end{aligned}$$

$$\begin{aligned}
\textcircled{\alpha}\ f\ (\mu_X(s)) &= (\alpha \circ T(f))\ (\mu_X(s)) \\
&= (\alpha \circ T(f) \circ \mu_X)\ s \\
&= (\alpha \circ \mu_A \circ T^2(f))\ s \\
&= (\alpha \circ T\alpha \circ T^2(f))\ s \\
&= (\alpha \circ T(\alpha \circ T(f))\ s \\
&= \textcircled{\alpha}\ s\ (\alpha \circ T(f)) \\
&= \underset{t \in s}{\textcircled{\alpha}}\ (\textcircled{\alpha}\ f)\ t
\end{aligned}$$

Proof of the construction of list algebras from monoids

This section provides the formal proofs in Agda that the algebra laws hold in the construction of list algebras from monoids in § 3.2.2. These proofs were used for formal verification of the theory.

```
unit′ : (α′ ∘ (M—η A′)) C.≡ C.id
unit′ {x} = begin
  α′ ∘ M—η A′ ⟨$⟩ x
≈⟨ refl ⟩
  x ● ε
≈⟨ proj₂ (Monoid.identity monoid) x ⟩
  x
∎

struct′ : ∀ {x} → (α′ ∘ M—μ A′) ⟨$⟩ x ≈ (α′ ∘ F₁ α′) ⟨$⟩ x
struct′ {[]} = refl
struct′ {[] ∷ x₁} =
  begin
    foldr _●_ ε (concat x₁)
  ≈⟨ struct′ {x₁} ⟩
    foldr _●_ ε (map (foldr _●_ ε) x₁)
  ≈⟨ sym (proj₁ (Monoid.identity monoid) _) ⟩
    ε ● foldr _●_ ε (map (foldr _●_ ε) x₁)
  ≈⟨ refl ⟩
    foldr _●_ ε (map (foldr _●_ ε) ([] ∷ x₁))
  ∎
struct′ {(x ∷ x₁) ∷ x₂} =
  begin
    x ● foldr _●_ ε (concat (x₁ ∷ x₂))
  ≈⟨ ●—cong refl ( struct′ {x₁ ∷ x₂}) ⟩
    x ● (foldr _●_ ε x₁ ● foldr _●_ ε (map (foldr _●_ ε) (x₂)))
  ≈⟨ sym (Monoid.assoc monoid _ _ _) ⟩
    x ● foldr _●_ ε x₁ ● foldr _●_ ε (map (foldr _●_ ε) (x₂))
  ∎
```

Parametrised big operators

This section shows that we can recover the definition of the parametrised big operator defined in § 3.2.6 from the non-parameterised big operator using the idea of an *exponential object* from category theory. An exponential object in a category with binary products is an object Z^Y together with a morphism *eval* $: (Z^Y \times Y) \to Z$ with some properties. In the category $\mathcal{S}etoids$, an exponential object Z^Y is the set of all functions $Y \to Z$ and the map *eval* is just the evaluation map sending the pair (f, y) to $f(y)$.

By applying the big operator for the T-algebra $(Y \to A, \alpha^Y)$ (where $\alpha^Y : T(Y \to A) \to (Y \to A)$) to the curried version, say f_1 of f of type $X \to (Y \to A)$, we get

$$\textcircled{\scriptsize α^{Y}} f_1 : TX \to (Y \to A)$$

whose uncurried version of type $TX \times Y$ yields $@_1 f$. We can do this similarly with $(X \to A, \alpha^X)$.

Commutative monads

In this section, we prove that

$$\underset{2}{@} (\underset{1}{@} f) \text{ and } \underset{1}{@} (\underset{2}{@} f)$$

are equal whenever the monad is commutative.

Recall the definitions of $@_1$ and $@_2$ from § 3.2.6: Then

$$\begin{aligned}
\underset{2}{@} (\underset{1}{@}\ f) &= \alpha \circ T(\underset{1}{@}\ f) \circ \mathit{st} \\
&= \alpha \circ T(\alpha \circ T(f) \circ \mathit{st}') \circ \mathit{st} \\
&= \alpha \circ T(\alpha) \circ TT(f) \circ T(\mathit{st}') \circ \mathit{st}
\end{aligned}$$

and

$$\begin{aligned}
\underset{1}{@} (\underset{2}{@}\ f) &= \alpha \circ T(\underset{2}{@}\ f) \circ \mathit{st}' \\
&= \alpha \circ T(\alpha \circ T(f) \circ \mathit{st}) \circ \mathit{st}' \\
&= \alpha \circ T(\alpha) \circ TT(f) \circ T(\mathit{st}) \circ \mathit{st}'
\end{aligned}$$

A categorical proof of the equality follows.

$$\begin{tikzcd}
TX \times TY \arrow[r, "st_{TX,Y}"] \arrow[d, "st'_{X,TY}"'] & T(TX \times Y) \arrow[r, "T(st'_{X,Y})"] \arrow[rrr, bend left, "T(@_1\ \ f)"] \arrow[d, phantom, "\textit{commutativity}"] & T^2(X \times Y) \arrow[r, "T^2(f)"] \arrow[d, "\mu_{X\times Y}"'] \arrow[dr, phantom, "\textit{naturality}"] & T^2A \arrow[r, "T(\alpha)"] \arrow[d, "\mu_A"] & TA \arrow[d, "\alpha"] \\
T(X \times TY) \arrow[r, "T(st_{X,Y})"] \arrow[rrrr, dr, bend right, "T(@_2\ \ f)"'] & T^2(X \times Y) \arrow[r, "\mu_{X\times Y}"] \arrow[rrd, "T^2(f)"'] & T(X \times Y) \arrow[r, "T(f)"] & TA \arrow[r, "\alpha"] & A \\
 & & & T^2A \arrow[r, "T(\alpha)"] \arrow[u, "\mu_A"] \arrow[ur, phantom, "\textit{eq. }(2.17)"] & TA \arrow[u, "\alpha"']
\end{tikzcd}$$

Computer Science Tripos – Part II – Project Proposal

Theory and implementation of a general framework for big operators in Agda

Stella Lau, Trinity College

Originator: Timothy Griffin

20 October 2016

Project Supervisor: Timothy Griffin & Marcelo Fiore

Director of Studies: Sean Holden & Frank Stajano

Project Overseers: Markus Kuhn & Peter Sewell

Introduction

In mathematics, we often use notation such as

$$\sum_{i=1}^{n} f(i)$$

to denote the use of a binary operator over a collection of values bracketed in some prescribed way that does not matter when the operator is associative. This notation is notably missing in its full generality from proof assistants and is not present at all in Agda, a dependently-typed language and proof assistant. A large body of mathematical theory exists to reason about big operations, some of which do not rely on any assumptions on the operator and others of which assume some properties of the operators such as an Abelian monoid structure or a semi-ring structure.

The goal of this project is to develop the theory for and implement a generic framework to reason about a variety of big operators in Agda, from simple boolean conjunction to a general *max arg* construction, indexed by arbitrary types. The project will extend and generalise previous work on big operator libraries using the more general framework of monad algebras [6].

Algebras over monads

We will abstractly regard big operators as monad algebras. The notion of monad here represents "container types". Examples include lists and generalisations such as sets, multisets, and trees. In category theory, a monad is an endofunctor (a functor that maps a category to itself) together with two natural transformations (functor morphisms) endowing the former with a monoid structure. A monad has the form (T, η, μ), with $\eta : \forall A, A \to T(A)$ and $\mu : \forall A, T^2(A) \to T(A)$, and obeys certain axioms.

We can think of T as both a type former, $T(A)$, and a way of polymorphically transforming functions, $T(f)$. For example, if $T(A) = \texttt{List}\ A$ and $T(f) = \texttt{map}\ f$, then $\eta(a) = [a]$ (constructing a list from a singleton) and $\mu(l) = \texttt{concat}\ l$ (like flatten, taking a list of lists and flattening them to a single list). A T-algebra (R, α) with $\alpha : T(R) \to R$ gives rise to a big operator from which we can derive interesting equations without many assumptions on T or α.

This project will further develop and formalise this theory and provide an implementation in Agda based on this general framework. The use of the library will then be demonstrated by providing some proofs in Agda. As a reach goal, we hope to end up with a library we can release to the public.

Starting point

Implementations of big operators exist in Isabelle [1] and Coq [2]. Isabelle's library for big operators is based on an effort to formalise set theory and uses sets instead of the more general lists. Coq's library [3] for uniformly iterated big operations is more similar to the intended approach of this project, with no restrictions on the underlying structure. We aim to use previous work by Leonhard Mackert from his Part III dissertation as a prototype to extend and refine using generalizations based on category theory.

Resources required

For this project I shall mainly use my own quad-core computer that runs Ubuntu. I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure. Backup will be to github and to an external hard disk. The MCS machines are on hand should my main machine suddenly fail. I require no other special resources.

Work to be done

The project breaks down into the following sub-projects:

1. Familiarisation with Agda, dependent types, and any surrounding software tools.
2. Learning the fundamentals of category theory and conducting a detailed study of previous work both in Cambridge and elsewhere on the underlying algebraic theory of big operators.
3. Conducting a detailed design of data structures representing index structures and operator structures aiming for generality (big operators indexed by arbitrary types and the library being useful for different kinds of types and operations). Structuring the code and module sensibly following the conventions used in the Agda standard library.

4. Implementing syntax definitions and a framework grounded in theory that can express indexes, range descriptions, and operations.

5. Proving and implementing lemmas to reason about big operators without any assumptions on the operator as well as with assumptions on structure (e.g. a monoid structure).

6. Evaluation using illustrative proofs and comparison with proofs without the library. Collating the evidence that will be included in the dissertation to demonstrate that the code behaves as required.

7. Writing the dissertation.

Success criteria

The project will be deemed a success if I have completed the following:

1. Developed and formalised theory for a general framework for big operators based on algebras of monads. This approach is more general than used in Isabelle or Coq and should allow us to get more equations over more kinds of monads in a more general way.

2. Designed and implemented a generic framework for big operators in Agda, based on the theory of algebras of monads, having the ability to represent and define big operators with different properties (such as associativity, commutativity, or distributivity) and monads of different types such as lists or sets.

3. Demonstrated the use of the library by proving various equivalences using big operators such as results on summations from *Concrete Mathematics* [4]. For instance:

$$\sum_{j\in J, k\in K} a_j b_k = \left(\sum_{j\in J} a_j\right)\left(\sum_{k\in K} b_k\right)$$

Possible extensions

If I achieve my main result early, I can try some of the following:

1. Instantiate and prove results using non-standard monads such as multisets, indexed containers, or binary trees (as opposed to "standard" monads like lists and sets).

2. Prove more complex theorems using more complex algebraic structures. For instance, proving the Cayley-Hamilton theorem or the Cauchy-Binet formula, which states that the determinant of a product of square matrices is equal to the product of their determinants for matrices with entries from any commutative ring.

3. Clean up the code and attempt to release it to the general Agda community.

4. Formally verify the correctness of the implementation.

5. Use reflection [7] to automate congruence in Agda to simplify proofs.

Evaluation

Some ways I can evaluate my implementation include the following:

1. Conducting unit testing and discussing their effect on confidence in the correctness of the implementation.
2. Discussing the complexity of proofs implemented using the library using metrics such as lines of code.
3. Discussing the readability of proofs implemented using the library by comparison to proofs without the library.
4. Comparing the generality, limitations, and structure of the implementation with that of other big operator libraries.
5. Discussing the extent to which the implementation conforms with Agda standards and is intuitive to use for Agda users.

Timetable

1. **Michaelmas weeks 2–4** Install Agda and learn how to use it. Practice writing small proofs and become familiarised with the standard library. Set up github repository and get backup strategy in place. Learn category theory by reading ahead in Part III course notes or reading a book. Study relevant literature and implementations such as Coq's `bigop`.
2. **Michaelmas weeks 5–6** Develop theory to represent big operations as algebras over monads. Design and implement basic prototype modules to represent big operators and monads based on this theory.
3. **Michaelmas weeks 7–8** Iterate on prototype and prove basic summation lemmas. Start drafting introduction and preparation sections of the dissertation.
4. **Michaelmas vacation** Finish implementing and testing a working prototype with modules required to prove summation lemmas. Prove some theorems with matrices if time allows. Start drafting the implementation section of the dissertation and finish the draft of the introduction and preparation sections.
5. **Lent weeks 0–2** Write progress report. Generate some proof examples using the library and evaluate its design. Plan to redesign, modify, and add parts as necessary.
6. **Lent weeks 3–5** Achieve a working project and tidy up code.
7. **Lent weeks 6–8** Attempt extensions. Have a very rough draft of the entire dissertation and get feedback.
8. **Easter vacation:** Incorporate feedback from reviewers. Finish writing dissertation fully. Ensure code is in a state ready for submission.
9. **Easter term 0–2:** Proof read dissertation.
10. **Easter term 3:** Submit dissertation.

References

[1] Paulson, Lawrence C. and Tobias Nipkow (1994). *Isabelle: a generic theorem prover.* Lecture notes in computer science 828. Berlin; New York: Springer. 321 pp.

[2] Huet, Gérard, Gilles Kahn, and Christine Paulin-Mohring (2015). *The Coq Proof Assistant: A Tutorial (version 8.4pl6).*

[3] Bertot, Yves et al. (2008). *Theorem Proving in Higher Order Logics: 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings,* chapter Canonical Big Operators, pages 86-101. Springer Berlin Heidelberg, Berlin, Heidelberg,

[4] Graham, Ronald L, Donald E. Knuth, and Oren Patahnik. (1994). *Concrete mathematics: a foundation for computer science.* 2nd ed. Reading, Mass: Addison-Wesley, pp. 21-62.

[5] Gustafsson, Daniel and Nicolas Pouillard (2014). *Foldable containers and dependent types.* `https://nicolaspouillard.fr/publis/explore-iso.pdf`.

[6] Griffin, Timothy G. and Marcelo Fiore (2016). *On the Algebras of Big Operators.* Unpublished manuscript.

[7] Walt, Paul van der and Wouter Swierstra (2013). "Engineering Proof by Reflection in Agda". In: *Implementation and Application of Functional Languages.* Ed. by Ralf Hinze. Lectures Notes in Computer Science 8241. Berlin; heiderlberg: Springer, pp. 157-173.