Leonhard Markert
Emmanuel College

Big operators in Agda

Part III project report · June 2015

Supervised by
Timothy Griffin and *Dominic Mulligan*

We present a library for expressing and proving conjectures involving “big operators” like $\sum_{i=0}^{n} f(i)$ in Agda, a dependently typed language and proof assistant. We argue that the essence of any such operation is encapsulated in a monoid, an algebraic structure with an identity and associativity law.

In addition to big operators, we formalise intervals of natural numbers, filters and matrices. To demonstrate the definitions and lemmas included in our library, we prove two variants of the Gauss formula as well as the Binomial theorem and the theorem “square matrices over a semiring form a semiring”.

Proforma

Name and College	Leonhard Markert, Emmanuel College
Project Title	Big operators in Agda
Examination	Computer Science Tripos, Part III (June 2015)
Word Count	11999 words[1]
Project Originators	Timothy Griffin, Dominic Mulligan and Leonhard Markert
Project Supervisors	Timothy Griffin and Dominic Mulligan

Declaration of Originality

I, Leonhard Markert of Emmanuel College, being a candidate for Part III 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	
Date	2nd June 2015

[1]Number of words in the main body of the report. This word count includes footnotes, but excludes code listings and appendices.

Contents

1 Introduction 1
1.1 Motivation . . . 2
1.2 Aims and contributions . . . 2
1.3 Overview . . . 3

2 Background 4
2.1 Agda basics . . . 4
2.1.1 Small types and functions . . . 5
2.1.2 The type hierarchy and universe polymorphism . . . 6
2.1.3 Dependent types and indexed type families . . . 8
2.1.4 Record types . . . 9
2.2 Predicates and relations . . . 10
2.2.1 Predicates . . . 10
2.2.2 Relations . . . 12
2.3 Provability and decidability . . . 13
2.3.1 Type inhabitation . . . 13
2.3.2 Decidability . . . 14
2.4 Equivalences and setoids . . . 15
2.4.1 Equivalences . . . 15
2.4.2 Setoids . . . 15
2.4.3 Equational reasoning . . . 16
2.5 Algebra . . . 17
2.5.1 Properties of binary operators . . . 17
2.5.2 Algebraic structures . . . 18

3 Implementation 20
3.1 Design . . . 20
3.2 Big operators . . . 21
3.2.1 Representing indices as lists . . . 22
3.2.2 Monoid structure . . . 22
3.2.3 Implementing big operators . . . 23
3.3 Intervals . . . 25
3.4 Filters . . . 26
3.5 Properties of big operators . . . 28
3.5.1 Monoid lemmas . . . 29
3.5.2 Commutative monoid lemmas . . . 29
3.5.3 "Semiring without one" lemmas . . . 30

3.5.4 Boolean algebra lemmas 30

4 Square matrices over semirings 31
4.1 Matrices 31
4.2 Definitions 33
4.3 Properties of matrix addition 35
4.4 Properties of matrix multiplication 37
4.5 Distributivity laws 43
4.6 Summary 45

5 Conclusions 46
5.1 Related work 47
5.2 Future work 48

Bibliography 49

A Predicate example 51

B Gauss formula 53
B.1 Gauss formula 53
B.2 Odd Gauss formula 54

C Binomial theorem 57
C.1 Pen-and-paper proof 57
C.2 Definitions 57
C.3 Lemmas 58
C.4 Proof 61

D Additional proofs 63
D.1 Arrow-pair isomorphism 63
D.2 Equality of left and right fold 64
D.3 Extensional equality with reducebig 65

Chapter 1

Introduction

"SUMS ARE EVERYWHERE": so begins the second chapter of *Concrete mathematics: a foundation for computer science* (Graham 1994, p. 21). Its authors are referring to the syntax used to express, for example, the "odd Gauss formula":

$$n^2 = \sum_{\substack{i=0 \\ i \text{ odd}}}^{2n} i \tag{1.1}$$

This notation is very concise. It allows us to specify a set of indices (all odd natural numbers between zero and $2n$ in this case); define an expression in which the index variable is locally bound (here the expression only consists of the bound variable i); and lift a binary operator (addition in this case) to an entire set.

Similar syntax exists for products ($\prod$), unions ($\bigcup$), conjunctions ($\bigwedge$) and other operations. Writing terms in this way often improves our intuition about the problem at hand. The following equalities, for example, are usually taken for granted in pen-and-paper mathematical reasoning:

$$\sum_i \sum_j a_{i,j} = \sum_j \sum_i a_{i,j} \qquad \prod_i \prod_j a_{i,j} = \prod_j \prod_i a_{i,j} \tag{1.2}$$

$$\prod_i a_i * b_i = \left(\prod_i a_i\right) * \left(\prod_i b_i\right) \qquad \bigcup_i a_i \cup b_i = \left(\bigcup_i a_i\right) \cup \left(\bigcup_i b_i\right) \tag{1.3}$$

$$\bigcup_i a \cap b_i = a \cap \left(\bigcup_i b_i\right) \qquad \bigwedge_i a \vee b_i = a \vee \left(\bigwedge_i b_i\right) \tag{1.4}$$

In formal mathematics, we collectively refer to these notational devices as "big operators" (Bertot et al. 2008). It can be shown that Equation (1.2) and Equation (1.3) hold for *any* operation that is both associative and commutative (see Section 3.5.2). This idea leads to a general theory of big operators, where the laws governing a big operator (e.g., $\prod$) are derived from the properties of its underlying operation (multiplication in this case).

Proof assistants like *Isabelle* (Paulson and Nipkow 1994), *Coq* (Huet, Kahn, and Paulin-Mohring 2015), or *Agda* (Norell 2009) simplify the development of formal proofs. Providing a notation for big operators in a proof assistant is an obvious way to extend the number of proofs that can be expressed naturally. Isabelle and Coq both have libraries that contain syntax definitions and lemmas for dealing with big operators (see Section 5.1).

1.1 Motivation

Our initial motivation for this project came from Timothy Griffin's Algebraic Path Problems course (L11) on modelling shortest-path and related problems algebraically.[1] Matrix multiplication is used to compute the solutions to path problems in this framework. And matrix multiplication, in turn, is usually defined in terms of a big operator:

$$(A\,B)_{r,c} = \sum_{i=0}^{n} A_{r,i}\,B_{i,c}$$

Big operators are also common in number theory, discrete probability, combinatorics and other areas of mathematics (Graham 1994). As an example of a proof with big operators, we can show that matrix multiplication is associative as follows:

$$\begin{aligned}
((A\,B)\,C)_{r,c} &\approx \sum_{i=0}^{n}\left(\sum_{j=0}^{n} A_{r,j}\,B_{j,i}\right) C_{i,c} && \text{by definition}\\
&\approx \sum_{i=0}^{n}\sum_{j=0}^{n}(A_{r,j}\,B_{j,i})C_{i,c} && \text{by right-distributivity } (\star)\\
&\approx \sum_{j=0}^{n}\sum_{i=0}^{n}(A_{r,j}\,B_{j,i})C_{i,c} && \text{swapping the outer sums } (\star)\\
&\approx \sum_{j=0}^{n}\sum_{i=0}^{n} A_{r,j}\,(B_{j,i}\,C_{i,c}) && \text{by associativity}\\
&\approx \sum_{j=0}^{n} A_{r,j} \sum_{i=0}^{n} B_{j,i}\,C_{i,c} && \text{by left-distributivity } (\star)\\
&\approx (A\,(B\,C))_{r,c} && \text{by definition}
\end{aligned}$$

The proof uses three big operator equalities (marked with $\star$).

1.2 Aims and contributions

Agda is well equipped for writing proofs that resemble pen-and-paper mathematics due to a technique called *equational reasoning* (see Section 2.4.3), but currently lacks a big operator library. The aim of this project was to implement syntax definitions and lemmas that allow Agda users to write proofs that involve big operators.

This aim has been achieved—for example, see Page 40 for a formal proof that matrix multiplication is associative using the same reasoning steps as shown above. Our library enables Agda users to utilise syntax and reasoning principles like in Equations (1.2) to (1.4) familiar from pen-and-paper mathematics in proofs involving big operators.

Chapter 4, Appendix B and Appendix C demonstrate the use of the reasoning principles and syntax provided by our library, which are discussed in Section 3.5.

[1]Gondran (2008) is the reference for the algebraic constructions used in L11.

The main contributions of this project are:

- We have proved and packaged reasoning principles for big operators based on the algebraic properties of their underlying binary operators (see Section 3.5) for use in equational Agda proofs.
- We have designed compositional syntax definitions for writing sums and other big operators, intervals of natural numbers and filters in Agda that mimic standard mathematical notation (see Sections 3.2 to 3.4).
- We provide a formal proof of the theorem “square matrices over a semiring again form a semiring” (Chapter 4), two identities attributed to Gauss (Appendix B) and a proof of the Binomial theorem (Appendix C) in Agda.

The Agda code and module structure of the implementation follow the same conventions as the Agda standard library.[2] We took care not to duplicate work and used definitions from the standard library wherever possible.

1.3 Overview

Chapter 2 is a tutorial-style introduction to Agda and dependent types in general. A detailed description of our library is given in **Chapter 3**. In **Chapter 4** we prove that square matrices over a semiring form a semiring, demonstrating the definitions and lemmas developed in this project. Finally in **Chapter 5** we discuss related work and ideas for future research.

The **Appendices** contain an extended example of a predicate (Appendix A) to reinforce the understanding of predicates developed in Section 2.2.1. We prove Equation (1.1) and one of its variants (Appendix B) and the Binomial theorem (Appendix C). In Appendix D we provide additional proofs for some claims made in the main body of the report.

[2]An overview of the standard library as a clickable source code file is presented here: `https://agda.github.io/agda-stdlib/README.html`

Chapter 2

Background

In this Chapter, we provide the necessary background for the remainder of the report to make it self-contained.

Section 2.1 introduces Agda by example. The next three Sections cover some fundamental notions in constructive higher-order logic: we introduce predicates and relations as they are defined in Agda (Section 2.2), discuss provability and decidability and how they correspond to type inhabitation (Section 2.3) and present setoids and the equational reasoning style of writing proofs (Section 2.4). In Section 2.5 we review some algebraic properties of binary operators and algebraic structures like monoids and semirings.

Code listings

Syntax highlighting is used to make the code listings in this report easier to read. Table 2.1 lists the different syntactic categories and how they are typeset in this report.[1]

Notation	*Role*
refl [] _::_	Datatype constructors
List _≡_ Pred	Datatype names
fold ∗−comm •−cong	Functions
x ys A f	Bound variables
open with where	Reserved keywords

Table 2.1: Colours used in Agda code listings.

2.1 Agda basics

This project was implemented in *Agda*, a functional programming language with a dependent type system. *Functional programming* is a declarative paradigm where computation proceeds by evaluating expressions (instead of, say, changing the state of an abstract machine.) In a *dependent* type system, types can contain (depend on) terms—examples of dependent types are given in Section 2.1.3 and Section 2.1.4.

[1]This report is a *literate Agda file*: every piece of code that is displayed in a grey code box is type checked. Some import statements and definitions are not displayed in the output document for brevity.

The advantage of a dependently typed language like Agda over non-dependently typed functional languages like *Haskell* (Marlow 2010) or *ML* (Milner 1997), is that the type system is more expressive: under the Curry-Howard correspondence, it also serves as a higher-order logic where formulae are encoded as types and terms inhabiting those types witness derivations (Howard 1980; Sørensen and Urzyczyn 2006). The disadvantage is that type inference is undecidable, so most terms need type annotations.

We now introduce the syntax of Agda. The following sections explain how Agda can be used to write theorems and check proofs.

2.1.1 Small types and functions

Truth values (Booleans) can be defined in Agda as follows:

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

We introduce a new type Bool with two constructors, true and false. Both construct elements of Bool, so that is the type we annotate them with (after the colon). It may come as a surprise that the type Bool itself needs an annotation, too. In Agda, the type of small types is called Set. Since Bool is a small type, we declare it to be of type Set. In Section 2.1.2 we introduce types that are not contained in Set.

The function not flips its Boolean argument:

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

This function takes a Bool and returns a Bool, so the type of the function as a whole is Bool → Bool. The function is defined by pattern matching: the result of the function is the term on the right-hand side of the equality sign if its input matches the left-hand side. Note that the patterns must cover all possible cases. All Agda functions must be *total*, that is, defined on all values of its argument types.

Agda identifiers can contain Unicode symbols, which makes it possible to use notation familiar from mathematics in Agda code. The function _∧_ computes the logical conjunction of its inputs:

```
_∧_ : Bool → _ → Bool
true  ∧ true  = true
_     ∧ _     = false
```

An underscore (the symbol "_") can be interpreted in three different ways in Agda, depending on the context of its use. This slightly contrived example covers them all:

In an identifier like _∧_, the underscores indicate where the function or type expects its arguments. This allows programmers to introduce new infix and mixfix operators (Danielsson and Norell 2011).

In place of a term or type an underscore stands for a value that is to be inferred by Agda. In the type of our function, Bool → _ → Bool, the underscore marks the type of the second argument to _∧_, which is easily resolved to Bool.

As a pattern the underscore matches anything. It acts like a fresh pattern variable that cannot be referred to. The definition of _∧_ can thus be read as "the conjunction of two Booleans is true if both arguments are true; in any other case, the result is false."

We now consider a type with more structure. A natural number is either zero, or the successor of some natural number. In Agda, this inductive definition is written as:

```
data ℕ : Set where
  zero  : ℕ
  suc   : ℕ → ℕ
```

Addition of natural numbers can then be defined as follows:

```
_+_ : ℕ → ℕ → ℕ
zero    + n = n
suc m   + n = suc (m + n)
```

Pattern variables like m and n are bound in the function body. They are substituted for the values passed to the function when it is evaluated.

Agda functions defined on inductive types must *terminate*. Since termination checking is undecidable in general, Agda checks whether the arguments to the recursive call are structurally smaller than the arguments to the caller as a safe syntactic approximation to termination.

This is clearly the case in the recursive case of _+_: the arguments to the caller are suc m and n whereas those passed used in the recursive call are m and n. Structurally, m is smaller than suc m so Agda can infer that the function terminates on all inputs. A formal definition of *structurally smaller* is given in Coquand (1992).

2.1.2 The type hierarchy and universe polymorphism

Every type in Agda resides somewhere in a type universe with countably infinite levels. This hierarchy of types exists in order to approximate the typing judgement Set : Set while avoiding logical inconsistency (Girard 1972).

Bool and ℕ are examples of small types, which is expressed in Agda as Bool ℕ : Set (note that we can give type declarations for terms of the same type in one line in this way.) Set is a synonym for Set 0, which is itself of type Set 1.[2] This gives rise to an infinite predicative hierarchy of types, which approximates Set : Set in the limit:

```
Bool ℕ ... : Set 0
     Set 0 : Set 1
     Set 1 : Set 2
     Set n : Set (lsuc n)
```

With the concept of universe levels in mind, we now look at how the Agda standard library defines the type of lists:

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

Before we explain this datatype declaration in detail, we consider an examples first. The list bools contains Boolean values. Here the carrier type of the list, A, is instantiated to Bool.

```
bools : List Bool
bools = true :: (false :: ((true ∧ false) :: []))
```

In the datatype declaration of List, the names a and A are left of the colon. They are global to the constructors of the datatype and called *parameters*. The parameter a is written in curly braces, which marks it as an *implicit* parameter. This means that Agda will try to infer its value if it is not given explicitly.

Bool has type Set 0, so Agda correctly infers that in the type of bools, a must be 0. If we wanted to be explicit, we could have written the declaration as bools : List {0} Bool instead.

Since our definition of List abstracts over the level of its carrier type, we can also build lists that live higher up in the type hierarchy. We might for example want to create a list of types:

```
types : List Set
types = Bool :: (((ℕ → ℕ) → Bool) :: [])
```

The carrier type of the list is instantiated as Set, which is itself of type Set 1. The value of the implicit parameter a is inferred as level 1. Note that the function type (ℕ → ℕ) → Bool is also a small type.

Lists defined in this way are *universe polymorphic*, meaning that the universe level at which any particular list resides depends on its parameters. Making a parameter or argument of type Level implicit is common practice in Agda. Most of the time, the type checker can infer universe levels without ambiguity.

[2] We use numbers 0, 1, 2 to denote universe levels for brevity here. In actual code, elements of the opaque type Level can only be constructed using the postulated functions lzero and lsuc.

2.1.3 Dependent types and indexed type families

We now turn to the classic example of a dependent datatype (or *indexed family of types*): fixed-length lists, or *vectors*:

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

The parameters (on the left-hand side of the colon in the type declaration) are the same as for List, a type level a and a type A. Note that we have not specified the type of a—Agda infers that it must be a Level. In addition to the parameters, the type Vec is *indexed* by a natural number, hence the ℕ on the right-hand side of the colon. While parameters are the same for all constructors, indices may differ: the constructor [] returns a zero-length vector, while _::_ takes an element and a vector of length n and returns a vector of length suc n.

The vector append function _++_ demonstrates how dependent types can be used to enforce invariants. It takes two vectors of length m and n, respectively, and returns a vector of length $m + n$. The fact that the type checker accepts the definition of this function means that the length of the vector that is returned always is the sum of the lengths of the input vector.

```
_++_ : {a : Level} {m : ℕ} {n : ℕ} → {A : Set a} →
       Vec A m → Vec A n → Vec A (m + n)
[]        ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

As another example, consider a function head which returns the first element of a vector. We would like to restrict the argument of the function to vectors containing at least one element. The type of the following function does just that:

```
head : ∀ {n a} {A : Set a} → Vec A (suc n) → A
head (x :: _) = x
```

This definition does not violate totality even though there is no pattern for the empty vector. The reason is that the constructor for the empty vector has type [] : Vec A zero. Agda knows that suc n and zero cannot be unified, so we do not have to (and indeed cannot) supply a pattern for the empty list, which is exactly what we wanted.

Lastly we consider the type of finite sets Fin, indexed by a natural number:

```
data Fin : ℕ → Set where
  zero  : {n : ℕ} → Fin (suc n)
  suc   : {n : ℕ} → Fin n → Fin (suc n)
```

One way to think about this type is that its value represents a natural number with its index as an upper bound. The type Fin n has exactly n different values, representing the numbers up to $n - 1$. Fin zero is empty; there is no way to construct a value of this type.

A bounded natural number can be used as an index into a vector. This lets us leverage the type system to eliminate out-of-bounds handling. Consider the (slightly contrived definition of a) function lookup, which takes a position bounded by n and a vector of length n and returns the element at this position:

```
lookup : ∀ {n a} {A : Set a} → Fin n → Vec A n → A
lookup {.zero} ()       []
lookup         zero     (x :: xs) = x
lookup         (suc n)  (x :: xs) = lookup n xs
```

There are several things to note here. Firstly, it shows that we can match against implicit arguments by position. Here we use the pattern {.zero} to match the argument n. The curly braces indicate that we are matching against an implicit argument; the dot before the pattern marks this pattern as the unique value of the correct type. It is unique in this case because the constructor of the empty vector [] : Vec A zero appears in the same pattern, which forces the unification of n with zero. Note that *dotted patterns* do not have to be implicit, and that implicit arguments can also be matched against by name.

Secondly, since n is unified with zero, the position (the first non-implicit argument) would have to be of type Fin zero. But there is no value of this type! Agda's type checker can infer this and allows us to match this impossible value with (), the *absurd pattern*. No right-hand side is given for absurd patterns because they are never matched.

2.1.4 Record types

The record keyword lets us bundle terms and types together in a convenient manner. The type of a record field can depend on the values of any other field preceding it in the definition. A dependent pair type (or *Sigma type*) can be defined like this:

```
record Σ {a b} (A : Set a) (B : A → Set b) : Set (a ⊔ b) where
  constructor _,_
  field
    fst  : A
    snd  : B fst
```

The level of a record is the least upper bound $a \sqcup b$ of it fields' levels. Giving a constructor is optional in general but required for pattern matching. It also makes defining new values less verbose—compare pair_0 and pair_1 in the next example.

In the type declaration of Σ, the name B is given to a function which takes a *value* of type A and returns a *type* at level b. The type of snd is defined as $B\ fst$, so it *depends* on the value of fst. That is why Σ is called a dependent pair.For now we will restrict ourselves to building non-dependent pairs, which means that we will ignore the A-typed parameter to B. We can use Σ to define non-dependent pairs like this:

```
_×_ : ∀ {a b} → Set a → Set b → Set _
A × B = Σ A (λ _ → B)
```

This definition allows us to type non-dependent pairs more naturally:

```
pair₀ pair₁ : ℕ × Bool -- instead of Σ ℕ (λ _ → Bool)
pair₁ = record { fst = zero ; snd = false }
pair₀ = zero , false
```

As a record type, Σ can be deconstructed in many ways. The name of the type and the name of the field, separated by a dot, can be used to access that field (see fst_0). Alternatively, open followed by the name of the type can be used to bring the names of the field accessors into scope (see fst_1). The fields themselves can be brought into scope by opening the type followed by the value whose fields we want to access (see fst_2). Note that "let ... in x" and "x where ..." can be used almost interchangeably. In the last example, we pattern match on the constructor of the type to extract its fields (see fst_3).

```
fst₀ fst₁ fst₂ fst₃ : ∀ {a b} {A : Set a} {B : Set b} → A × B → A
fst₀ = Σ.fst
fst₁ = let open Σ in fst
fst₂ p = fst where open Σ p
fst₃ (x , _) = x
```

2.2 Predicates and relations

In this Section, we will see how predicates and relations are expressed in a dependent type system by example. We will then introduce the notion of *constructive logic* and how it relates to dependently typed programs.

From this point on, we will call some functions "proofs" and some types "theorems". The justification for this lies in the Curry-Howard correspondence, for which we will provide some intuition in this Chapter and the next. For a mathematically exact and systematic take on the Curry-Howard correspondence, see (Sørensen and Urzyczyn 2006).

2.2.1 Predicates

A predicate expresses some property of a term. The type of a predicate P over a type A is P : A → Set. The value of a predicate P x can be thought of as the *evidence* that P holds for x : A. We will look at a predicate Even : ℕ → Set as a warm-up, followed by a discussion of propositional equality, _≡_. For a discussion of a more involved predicate, Collatz : ℕ → Set, see Appendix A.

Even n provides evidence that n is an even number. One way of defining this predicate is by stating that any even number is either equal to zero, or it is the successor of the successor of an even number:

```
data Even : ℕ → Set where
  zero−even  : Even zero
  ss−even    : {n : ℕ} → Even n → Even (suc (suc n))
```

Using this definition, we can now provide evidence that zero and four are even numbers:

```
Even‿0 : Even 0
Even‿0 = zero−even

Even‿4 : Even 4
Even‿4 = ss−even (ss−even zero−even)
```

Since for some n : ℕ, Even n is a datatype, the evidence it represents can be analysed by pattern matching in proofs.

Next we look at *propositional equality*, written as _≡_ in Agda. The parameterised predicate _≡_ x expresses the property of "being equal to x". Two elements of the same type are propositionally equal if they can be shown to reduce to the same value.

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

The parameterised predicate _≡_ has only one constructor called refl.In order to create an inhabitant of the propositional equality type, we *must* use this constructor. It requires that its two arguments have the same value. Therefore, in order to obtain an inhabitant of $x \equiv y$, x and y must be shown to reduce to the same value.

Evaluating a function with equal arguments should always yield equal results. This property of a homogeneous relation is called *congruence*, and it can be proved for any single argument function f as follows:

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

Let us consider this proof step by step. We match the argument of type $x \equiv y$ with its only constructor, refl. This tells the type checker that x and y must be equal as refl : $x \equiv x$, and replaces all occurrences of y by x for this clause. To make this clearer we also pattern match against the implicit parameters x and y. Argument x is simply matched against a variable x. The dotted pattern for y is revealing: since the pattern refl forces x and y to be equal, the unique value that y can take is x.

This pattern match against $x \equiv y$ also has the effect of *rewriting* the type of the result expected on the right-hand side of the equals sign to $f\ x \equiv f\ x$. But as $f\ x$ and $f\ x$ are trivially equal, we can close the prove by simply using refl.

2.2.2 Relations

Usually in mathematics a relation between two sets is defined as a subset of the Cartesian product of the two sets. In a dependent type theory, a binary relation between types A : Set and B : Set has the type A → B → Set. It is not hard to show that this type is isomorphic to A × B → Set (see Appendix D.1 for a formal proof). Note that relations and predicates are closely related: a relation can be thought of as a predicate abstracted over some argument, and any relation can be applied to one argument to give a predicate.

We will restrict our attention to the special case of relations between inhabitants of the same type, called *homogeneous* relations. Formally, we can define predicates and homogeneous relations with an explicit universe parameter ℓ as follows:

```
Pred : ∀ {a} → Set a → (ℓ : Level) → Set (a ⊔ lsuc ℓ)
Pred A ℓ = A → Set ℓ

Rel : ∀ {a} → Set a → (ℓ : Level) → Set (a ⊔ lsuc ℓ)
Rel A ℓ = A → A → Set ℓ
```

The evenness predicate from the previous Section can now be typed as Even : Pred ℕ lzero. As an example, *Divisibility* is a familiar relation with a straightforward definition in Agda. It uses multiplication, so we define that first:

```
_*_ : ℕ → ℕ → ℕ
zero    * n = zero
suc m  * n = n + m * n
```

Now we can give a definition for the divisibility relation, which translates to “m divides n if you can provide a q such that $n \equiv qm$”.

```
data _|_ : ℕ → ℕ → Set {- equivalently, Rel ℕ lzero -} where
  divides : {m n : ℕ} (q : ℕ) (eq : n ≡ q * m) → m | n
```

The following proof demonstrates how this relation can be instantiated. It shows that 1 divides any natural number n:

```
1−divides−any : ∀ n → 1 | n
1−divides−any n = divides {1} {n} n n≡n*1
  where
    n≡n*1 : ∀ {n} → n ≡ n * 1
    n≡n*1 {zero}   = refl
    n≡n*1 {suc n}  = cong suc n≡n*1
```

The equality $n \equiv n * 1$ may seem rather obvious, and yet we need to prove it separately. This is because we defined multiplication by induction on its first parameter, so $1 * n$ normalises to n + zero but $n * 1$ cannot be evaluated further—it is “stuck”.

n≡n∗1 is a proof of the required equality by induction. The base case is n = zero. By the definition of _∗_, zero ∗ 1 = zero and the equality holds. In the inductive step, we need to show that suc n ≡ suc n ∗ 1. The right-hand side evaluates to 1 + n ∗ 1, which in turn evaluates to suc (n ∗ 1). The inductive hypothesis, n≡n∗1 {n} proves that n ≡ n ∗ 1. Our goal in the inductive step is show that suc n ≡ suc (n ∗ 1). The latter follows from the former by congruence.

2.3 Provability and decidability

In this Section, we make the relationship between Agda's type system and constructive logic more explicit, using types, predicates and relations from the previous Section as examples.

2.3.1 Type inhabitation

We proved in the previous Section that four is an even number by explicitly constructing a term of type Even 4. The fact the we were able to define a term of this type means that the type is *inhabited*, that is, it is a type with least one element.

Type inhabitation translates to *provability* in the constructive logic corresponding to Agda's type system: a type is shown to be inhabited if a term of that type can be given; in a constructive logic, a proposition is considered true when a constructive proof can be given for that proposition.

A proof of classical logic is a constructive proof if it does not use the law of excluded middle ($A \lor \neg A$) or proof by contradiction / double negation elimination ($\neg\neg A \to A$). The law of contradiction ($(A \to B) \to (A \to \neg B) \to \neg A$) and *ex falso quodlibet* ($\neg A \to A \to B$) are allowed.

Let us consider the following definition:

```
data ⊥ : Set where
```

The type ⊥ (pronounced "bottom") has no constructors, yet without termination checking we could define a function that just calls itself to inhabit ⊥. But this requires general recursion, which is not allowed in Agda because of the termination requirement. Therefore ⊥ has no inhabitants, which is why it is often called the *empty type*.

A type with no elements is useless from a computational perspective, but it makes for a useful definition of emptiness (the opposite of inhabitation). We claimed that it is impossible to create an element of type ⊥. But *ex falso quodlibet* means that we can derive anything, even ⊥, from an absurd assumption:

```
¬Even‿1 : Even 1 → ⊥
¬Even‿1 ()
```

Here the *only* pattern is an absurd pattern. The number one is neither zero nor the successor of the successor of any natural number, so an argument of type Even 1 is absurd.

We do not need to supply a right-hand side of the definition because an argument of type Even 1 can never be given.

We are allowed to give this undefined return value any type we like. Setting it to ⊥ carries a special meaning: because of termination checking, only functions whose argument type is empty can return the empty type, so the fact that we can define a function $A \to \bot$ for some type A means that A must be empty.

The following definition is simply a shortcut for the type of empty types:

```
¬_ : ∀ {p} → Set p → Set p
¬ P = P → ⊥
```

This lets us write the type of ¬Even‿1 more succinctly as ¬ Even 1.

2.3.2 Decidability

One question we may ask is whether there exists a terminating decision procedure for a given relation or predicate. In the case of a predicate P : A → Set, a decision procedure would be a function which for any argument x : A returns either an inhabitant of type P x (evidence that the predicate holds) or an inhabitant of type ¬ P x (evidence that the predicate does not hold). We can capture decidability in a type as follows:

```
data Dec {p} (P : Set p) : Set p where
  yes  : (p   :   P) → Dec P
  no   : (¬p  : ¬ P) → Dec P
```

For example, in order to show that the predicate Even is decidable on all natural numbers, we can define a function of type ∀ n → Dec (Even n):

```
even : ∀ n → Dec (Even n)
even 0  = yes zero−even
even 1  = no (λ ())
even (suc (suc n)) with even n
... | yes p   = yes (ss−even p)
... | no ¬p  = no (ss−odd ¬p)
  where
    ss−odd : ∀ {n} → ¬ Even n → ¬ Even (suc (suc n))
    ss−odd ¬ps (ss−even p) = ¬ps p
```

We have already covered the two base cases before: zero is clearly even and one is clearly not. The induction step is where things get interesting. Using a with-clause, we pattern match on whether n is even. If yes, then we can easily construct a proof of Even (suc (suc n)) by applying ss−even to the proof of Even n.

Otherwise, we build a proof of ¬ Even (suc (suc n)) from a element of ¬ Even n using ss−odd. Since ¬ A is just an abbreviation for $A \to \bot$, the type of ss−odd can also be written as ∀ {n} → (Even n → ⊥) → Even (suc (suc n)) → ⊥. The given pattern matches ¬ps : Even n → ⊥ and p : Even n. All we need to do to derive a contradiction is to apply ¬ps to p.

2.4 Equivalences and setoids

The main use case of the Bigop library is to prove equalities like the Binomial theorem (Graham 1994, Equation 5.13, page 163):

$$(1 + x)^n = \sum_{k=0}^{n} \binom{n}{k} x^k$$

In dependently typed languages, we often use the more general notions of equivalence and setoid in place of equality. These will be discussed in this Section.

2.4.1 Equivalences

A relation _≈_ is an equivalence if it is *reflexive*, *symmetric* and *transitive*. The IsEquivalence record bundles these three properties together:

```
record IsEquivalence {a ℓ} {A : Set a} (_≈_ : Rel A ℓ) : Set (a ⊔ ℓ) where
  field
    refl  : ∀ {x} → x ≈ x
    sym   : ∀ {x y} → x ≈ y → y ≈ x
    trans : ∀ {x y z} → x ≈ y → y ≈ z → x ≈ z

  reflexive : ∀ {x y} → x ≡ y → x ≈ y
  reflexive P.refl = refl
```

Here, reflexive is not a field but a proof that is brought into scope when IsEquivalence is opened. It shows that propositional equality _≡_ implies any other equivalence. In other words, reflexive shows that _≡_ is the smallest equivalence in Agda.

2.4.2 Setoids

A *setoid* packages a type, called the *carrier*, with a relation _≈_ defined on that type and a proof that this relation is an equivalence.

```
record Setoid c ℓ : Set (lsuc (c ⊔ ℓ)) where
  field
    Carrier       : Set c
    _≈_           : Rel Carrier ℓ
    isEquivalence : IsEquivalence _≈_
```

Setoids can be used to define quotients. For example, we could represent non-negative rational numbers as the setoid with carrier type ℕ × ℕ and equivalence relation _≈_ defined as $p , q \approx p' , q'$ if $p * q' \equiv p' * q$. Here the quotient _≈_ partitions the domain ℕ × ℕ into equivalence classes of pairs of natural numbers representing the same rational numbers.

2.4.3 Equational reasoning

Any setoid gives rise to a preorder, which consists of a carrier type and a relation with a reflexive and transitive law. This preorder can be used for *equational reasoning*, which provides syntactic sugar for applying the transitivity law. It aims to make Agda proofs look more like pen-and-paper proofs and will be used extensively in the next chapters.

As an example we take the setoid whose carrier is $\mathbb{N}$ with propositional equality $\_\equiv\_$ as the equivalence relation, and prove $(p * q) * r \equiv q * (p * r)$ in two different ways: first using transitivity explicitly, and then using equational reasoning.

The proofs assume that we have already shown that multiplication is commutative and associative, so the following are given:

$$\begin{aligned} \mathsf{*\!-\!comm} &: (m\, n : \mathbb{N}) \to m * n \equiv n * m \\ \mathsf{*\!-\!assoc} &: (m\, n\, o : \mathbb{N}) \to (m * n) * o \equiv m * (n * o) \end{aligned}$$

Additionally we have the transitivity law and congruence for binary functions, both parameterised over $\forall\, \{a\}\, \{A : \mathsf{Set}\; a\}$:

$$\begin{aligned} \mathsf{P.trans} &: \{x\, y\, z : A\} \to x \equiv y \to y \equiv z \to x \equiv z \\ \mathsf{P.cong_2} &: \{x\, x'\, y\, y' : \mathrm{A}\} \to (f : A \to A \to A) \to x \equiv x' \to y \equiv y' \to f\; x\; y \equiv f\; x'\; y' \end{aligned}$$

The underlying reasoning is the same for both proofs: we first show $(p * q) * r \equiv (q * p) * r$. It follows from $p * q \equiv q * p$ (by commutativity), $r \equiv r$ (by reflexivity) and congruence of $\_*\_$. Then we prove $(q * p) * r \equiv q * (p * r)$, which is a direct consequence of the associativity rule. Using the lemmas specified above, the two steps can be written like this:

$$\begin{aligned} \mathsf{P.cong_2}\; \_*\_\; (\mathsf{*\!-\!comm}\; p\; q)\; \mathsf{P.refl} &: (p * q) * r \equiv (q * p) * r \\ \mathsf{*\!-\!assoc}\; q\; p\; r &: (q * p) * r \equiv q * (p * r) \end{aligned}$$

The initial equation is proved by transitivity which links the two steps together. Using normal function application syntax to apply transitivity, we get the following proof:

```
equiv₀ : (p q r : ℕ) → (p * q) * r ≡ q * (p * r)
equiv₀ p q r = P.trans (P.cong₂ _*_ (*-comm p q) P.refl) (*-assoc q p r)
```

With equational reasoning it looks like this:

```
equiv₁ : (p q r : ℕ) → (p * q) * r ≡ q * (p * r)
equiv₁ p q r =
  begin
    (p *  q) * r  ≡⟨ P.cong₂ _*_ (*-comm p q) P.refl ⟩
    (q *  p) * r  ≡⟨ *-assoc q p r ⟩
    q * (p  * r)
  ∎
```

The proof starts with begin_ followed by the left-hand side of the equivalence we are trying to prove. It ends with the right-hand side of the equivalence followed by _∎ (which is meant to resemble the "q.e.d." symbol at the end of a proof). Intermediate steps are linked using _≡⟨_⟩_; the term in angle brackets provides the justification. Transitivity is applied implicitly. Note that there is nothing special about begin_, _≡⟨_⟩_ and _∎—they are defined in the standard library like any other mixfix operator.

Which proof style one prefers is a matter of taste. Equational reasoning is more verbose—equiv_1 spans seven lines compared to equiv_0's two—but it makes intermediate steps explicit, which improves readability.

In general, we may choose an equivalence _≈_ other than propositional equality for our setoid. We can then freely mix steps using that equivalence relation _≈⟨_⟩_ and steps using propositional equality _≡⟨_⟩_ in an equational reasoning-style proof. Proving intermediate steps by propositional equality is allowed because by reflexivity (see Section 2.4.1), $x \equiv y \to x \approx y$ for *any* equivalence relation _≈_.

2.5 Algebra

In this Section, we review some properties of binary operators and define monoids, commutative monoids and semirings in terms of those properties.

2.5.1 Properties of binary operators

A binary operator _⊗_ may have any of the following properties:

Associativity. $(a \otimes b) \otimes c \equiv a \otimes (b \otimes c)$. The order in which subterms are evaluated has no bearing on the result. If an operator is known to be associative, terms consisting of multiple applications of that operator are usually written without parentheses: $(a \otimes b) \otimes c \equiv a \otimes (b \otimes c) \equiv a \otimes b \otimes c$. In Algebra.FunctionProperties, a standard library module, associativity of an operator _•_ with respect to some relation _≈_ is defined as follows:

```
Associative : ∀ {a ℓ} {A : Set a} (_≈_ : Rel A ℓ) → (A → A → A) → Set _
Associative _≈_ _•_ = ∀ x y z → ((x • y) • z) ≈ (x • (y • z))
```

Identity (or unit). $1 \otimes a \equiv a$, $a \otimes 1 \equiv a$. Element 1 is called the left- or right-identity of _⊗_ in the first and second equation, respectively. In Agda's standard library, this property is encoded as a pair of predicates, LeftIdentity and RightIdentity:

```
LeftIdentity : ∀ {a ℓ} {A : Set a} (_≈_ : Rel A ℓ) → A → (A → A → A) → Set _
LeftIdentity _≈_ e _•_ = ∀ x → (e • x) ≈ x

RightIdentity : ∀ {a ℓ} {A : Set a} (_≈_ : Rel A ℓ) → A → (A → A → A) → Set _
RightIdentity _≈_ e _•_ = ∀ x → (x • e) ≈ x
```

Identity uses the non-dependent pair type _×_ to bundle the left- and right-identity laws together:

```
Identity : ∀ {a ℓ} {A : Set a} (_≈_ : Rel A ℓ) → A → (A → A → A) → Set _
Identity _≈_ e • = LeftIdentity _≈_ e • × RightIdentity _≈_ e •
```

Zero (or annihilator). $0 \otimes a \equiv 0$, $a \otimes 0 \equiv 0$. The value 0 is the left- or right-identity of _⊗_ in the first and second equation, respectively. This property is again encoded as pair of predicates in the same way as Identity. The left zero property is written as follows (RightZero is similar, and Zero is simply the pair of the two properties):

```
LeftZero : ∀ {a ℓ} {A : Set a} (_≈_ : Rel A ℓ) → A → (A → A → A) → Set _
LeftZero _≈_ z _•_ = ∀ x → (z • x) ≈ z
```

Commutativity. $a \otimes b \equiv b \otimes a$. Reordering operands does not change the result.

```
Commutative : ∀ {a ℓ} {A : Set a} (_≈_ : Rel A ℓ) → (A → A → A) → Set _
Commutative _≈_ _•_ = ∀ x y → (_•_ x y) ≈ (_•_ y x)
```

Binary operators may also interact in certain ways. If we add an operator _⊕_, we may, for example, get a distributive property:

Distributivity. $a \otimes (x \oplus y) \equiv (a \otimes x) \oplus (a \otimes y)$, $(x \oplus y) \otimes a \equiv (x \otimes a) \oplus (y \otimes a)$. We say that _⊗_ left- or right-distributes over _⊕_, respectively. Left-distributivity is encoded in Agda as follows:

```
DistributesOverˡ : ∀ {a ℓ} {A : Set a} (_≈_ : Rel A ℓ) →
                   (A → A → A) → (A → A → A) → Set _
DistributesOverˡ _≈_ _*_ _+_ =
  ∀ x y z → (x * (y + z)) ≈ ((x * y) + (x * z))
```

2.5.2 Algebraic structures

Certain combinations of the properties described in the previous subsection arise often, so for convenience, they are given names.

A *semigroup* packages a carrier type together with an associative operation _⊗_. If the operation has an identity ε, the structure is called a *monoid*. In a *commutative monoid*, the operation is also commutative. Two examples for commutative monoids over the natural numbers are addition and with the number zero as the identity element, and multiplication with the number one as its identity:

$$\begin{array}{ccc} (a+b)+c = a+(b+c) & a+b=b+a & a+0=0 \\ (a*b)*c = a*(b*c) & a*b=b*a & a*1=1 \end{array}$$

These two commutative monoids give rise to the big operators $\sum$ and $\prod$, respectively.

Given a monoid over _⊗_ and a commutative monoid over _⊕_, if the ⊕-identity (this element is called 0# in the standard library definition) is a zero for _⊗_ and _⊗_ distributes over _⊕_ we call the composite structure a *semiring*.

In the Agda standard library, the definitions of algebraic structures are split into two records, one containing the *properties* and the other the *data* of the structure.

Semigroups. The complete definition of a semigroup in Agda's standard library consists of the record types IsSemigroup and Semigroup:

```
record IsSemigroup {a ℓ} {A : Set a} (_≈_ : Rel A ℓ)
                   (_•_ : A → A → A) : Set (a ⊔ ℓ) where
open FunctionProperties _≈_
field
  isEquivalence  : IsEquivalence _≈_
  assoc          : Associative _•_
  •–cong         : ∀ {x x′ y y′ : A} → x ≈ x′ → y ≈ y′ → (x • y) ≈ (x′ • y′)
```

IsSemigroup encodes the properties of a semigroup. A is the carrier type and _≈_ an equivalence relation over this type. The properties, associativity (the predicate Associative is defined in the previous Section) and congruence, are instantiated with respect to that equivalence relation.

```
record Semigroup c ℓ : Set (suc (c ⊔ ℓ)) where
  field
    Carrier      : Set c
    _≈_          : Rel Carrier ℓ
    _•_          : Carrier → Carrier → Carrier
    isSemigroup  : IsSemigroup _≈_ _•_
```

The Semigroup record packages a Carrier type, a relation _≈_ and a binary operator _•_ together with the record containing the proofs that they satisfy the semigroup laws, isSemigroup.

Monoids. The definition of the Monoid record contains a field ε in addition to those already present in Semigroup. IsMonoid contains two fields, isSemigroup : IsSemigroup ≈ • and identity : Identity ε •. That is, in addition to being a semigroup, the structure must have an identity element ε.

Commutative monoids. CommutativeMonoid contains the same data as Monoid: an equivalence relation, an operator, and an identity. IsCommutativeMonoid extends IsSemigroup in as similar way as IsMonoid by adding an identity and a commutativity law.

Semirings "without one". SemiringWithoutOne is a structure almost like a semiring, except that it does not have a multiplicative identity.Its data contains *two* binary operators _+_ and _∗_ and a special element 0#.

Chapter 3

Implementation

In this Chapter we discuss the design and implementation of our big operator library. We formalise three independent concepts: big operators, intervals of natural numbers and filters. In combination, they allow for a large number of proofs involving big operators to be written in Agda. We will see how the expression

$$\forall\, n \to \Sigma[\, i \leftarrow 0 \dots n + n \parallel \mathsf{odd} \,]\; i \equiv n * n$$

is assembled from the syntax definition for big operators (Σ[_←_]_), intervals (_…_) and filters (_∥_):

Big operators. The module Bigop.Core defines an evaluation function and syntax for big operators. The submodules of Bigop.Properties contain lemmas about big operators on different algebraic structures such as monoids and semirings.

Intervals. Bigop.Interval contains functions for creating sequences of natural numbers and lemmas about those functions.

Filters. Bigop.Filter defines a function which filters a list based on a decidable predicate. The directory of the same name contains syntax definitions that help write equational reasoning proofs with predicates (Bigop.Filter.PredicateReasoning), definitions of the decidable predicates Even and Odd (Bigop.Filter.Predicates) and general lemmas about filters (Bigop.Filter.Properties).

The project's source code consists of 1,822 lines of code in 21 files. The module structure and naming follow the conventions used in the Agda standard library.

3.1 Design

Our goal in this project was to produce an Agda library for reasoning about big operators. We aimed to provide definitions and lemmas that abstracted over the particular operator being iterated. In several prototypes, we explored the design space for such a library. Two related questions had to be answered:

- What is the weakest algebraic structure that we can sensibly define a big operator on?

- How should the domain of indices of a big operator be represented?

The representation of the index domain and the minimal requirements on the algebraic structure depend on each other: the weaker the structure of the domain representation, the stronger the algebra has to be and vice versa. For example, the difference between lists and multisets is that lists are ordered. But in order to compute the iterated big operator over a multiset, the elements of the carrier must be combined using the underlying binary operator in *some* order, which in this case is arbitrary.

To get a well-defined result, the operator must consequently be immune to a re-ordering of its operands, in other words: removing the order from the domain adds commutativity to the properties required of the underlying operator. In Section 3.2.1 and Section 3.2.2 we argue that at least an identity and associativity law (that is, a monoid) is required, and that the appropriate index domain representation in this case is a list.

We built prototypes of alternative implementations, including a small finite sets library to represent the index domain (which added much complexity without tangible benefits) and an expression language for big operators—this idea may be worth reconsidering as part of an automated solver for big operator expressions (see Section 5.2). But for the use case of this library, representing a big operator by the result it evaluates to, and proving lemmas with respect to the underlying structure's equivalence relation turned out to be the best option (see Section 3.2.3).

3.2 Big operators

In this Section, we will see how big operators are evaluated using the Bigop module. We discuss why lists were chosen to represent indices, and why the binary operator must possess an identity and associativity law.

Nomenclature

In this report, a *big operator* is defined by

- an *underlying structure* at least containing a *carrier* type, a binary operator $\_\oplus\_$ and an element of the carrier, ε;
- a list of indices *is*; and
- a function f from the type of indices into the carrier.

The reasons for choosing this particular representation will be discussed in this section. We write the big operator corresponding to the structure described above as

$$\bigoplus_{i \leftarrow is} f(i)$$

Here $i \leftarrow is$ indicates that i ranges over *is*, the list of indices.

3.2.1 Representing indices as lists

Often in mathematical notation, the domain of a big operator expression is written using set notation. For this project, we decided to use lists instead for specifying the domain of a variable for the following reasons:

- Lists are *ordered*. With unordered sets, the evaluation of big operators on non-commutative binary operators is not well-defined.

  As an example, consider some arbitrary binary operator _⊕_. Then $\bigoplus_{i \in \{1,2\}} a_i$ could evaluate to either $a_1 \oplus a_2$ or $a_2 \oplus a_1$ since $\{1, 2\} = \{2, 1\}$. But if _⊕_ is non-commutative with $a_1 \oplus a_2 \neq a_2 \oplus a_1$, then the two results are not equal.

  Using lists to specify the domain of i, the problem evaporates: $\bigoplus_{i \leftarrow [1,2]} a_i$ (note the square brackets indicating that the domain of indices is a list) evaluates to $a_1 \oplus a_2$ only, so the result of the big operator expression is well-defined.

- Lists are well supported in Agda and commonly used: the standard library contains over 2000 lines of auxiliary definitions and lemmas about lists.

3.2.2 Monoid structure

In this Subsection we argue that a binary operator used to build a well-defined big operator must at least possess an identity element and satisfy associativity.

Identity

The evaluation function for big operators, fold (defined in Section 3.2.3), must be total like any other Agda function: it must accept any value in the domain of its argument types. For the list that the operator is being iterated over, one special value is [], the empty list.

What should a big operator iterated over an empty collection of indices evaluate to? By convention, for any binary operator _⊕_,

$$\bigoplus_{i \leftarrow []} f(i) \oplus x = x \qquad \text{and} \qquad x \oplus \bigoplus_{i \leftarrow []} f(i) = x$$

These two equations are exactly the left- and right-identity laws.

In order to simultaneously be able to compute any big operator over an list and enforce the intuition that it should behave as an identity, our library requires the binary operator to possess an identity element ε. The evaluation of big operators is defined such that it returns ε when the collection of indices the big operator being evaluated on is empty:

$$\bigoplus_{i \leftarrow []} f(i) = \varepsilon$$

Associativity

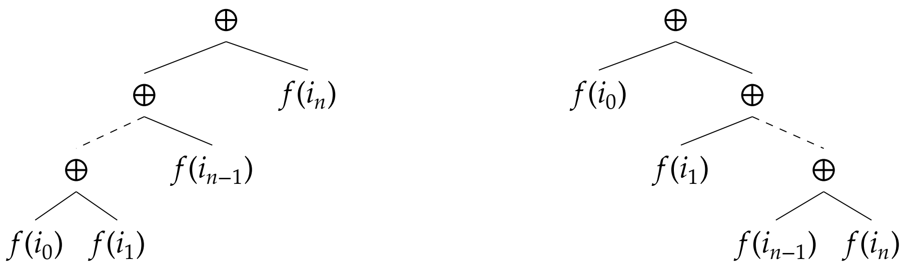
Assuming that i ranges over some non-empty list of indices $i_0, i_1, ..., i_{n-1}, i_n$, the big operator expression $\bigoplus_i f(i)$ is just an abbreviation for

$$f(i_0) \oplus f(i_1) \oplus \cdots \oplus f(i_{n-1}) \oplus f(i_n)$$

Without any further information about _⊕_, this string of symbols represents a *family* of expressions due to the lack of parentheses. Two members of that family are the left and right fold over the index list with f applied to each element:

$$(\cdots(f(i_0) \oplus f(i_1))\cdots \oplus f(i_{n-1})) \oplus f(i_n) \qquad \text{(left fold)}$$
$$f(i_0) \oplus (f(i_1) \oplus \cdots(f(i_{n-1} \oplus f(i_n)))\cdots) \qquad \text{(right fold)}$$

The expression tree notation clearly shows the difference between the two folds (left fold on the left, right fold on the right):



In order to actually compute the expression, we must decide on an order in which to add the terms. Unfortunately, the result of evaluating the left and right fold may differ. Our solution to this problem is to require _⊕_ to be an associative operation: in this case, all interpretations of the string of symbols representing the expanded big operator evaluate to the same value. Note that this does not resolve the ambiguity of which expression tree the string represents—it just means that any expression tree in which all elements appear in the same left-to-right order compute the same value, and are therefore propositionally equal as terms (see Appendix D.2 for a formal proof).

3.2.3 Implementing big operators

Recall from Section 2.5 the definition of a monoid in Agda:

```
record Monoid c ℓ : Set (suc (c ⊔ ℓ)) where
  field
    Carrier   : Set c
    _≈_       : Rel Carrier ℓ
    _•_       : Carrier → Carrier → Carrier
    ε         : Carrier
    isMonoid  : IsMonoid _≈_ _•_ ε
```

The record isMonoid contains proofs that $\_\approx\_$ is an equivalence relation, $\_\bullet\_$ is associative and congruent with respect to $\_\approx\_$ and ε is the identity for $\_\bullet\_$ (all with respect to $\_\approx\_$). One core idea of this project is that any monoid exactly specifies a big operator (see Section 3.2.2) as follows:

- The binary operator $\_\bullet\_$ gives us a way to combine elements of the monoid's carrier type. The associativity law guarantees that bracketing does not matter when we combine more than two elements. This means that left and right folds using $\_\bullet\_$ are equivalent.
- The identity element ε can be used as the result of a big operator expression over an empty index list. By the identity law, this makes any big operator expression over an empty index list behave as expected.

Given any monoid M, we can bring its fields into scope:

```
open Monoid M
```

Using the carrier type, the monoid's binary operator and identity element, we can then define the function crush which reduces a list to an element of the carrier type using a fold over that list:[1]

```
crush : List Carrier → Carrier
crush = foldr _•_ ε
```

crush xs defined over $\_\oplus\_$ computes $\bigoplus_{x \leftarrow xs} x$. The function itself is just an application of foldr, a right-fold over lists containing elements of the carrier type (as discussed in Section 3.2.1, the choice of fold is arbitrary here since the operator is associative: we could just as well have defined crush in terms of the left-fold function foldl). foldr returns its second argument (ε in this case) if the list passed to it is empty; otherwise it combines the list elements using its first argument, a binary operator ($\_\bullet\_$). The type of foldr specialised to our use case is:

$$\mathsf{foldr} : (Carrier \to Carrier \to Carrier) \to Carrier \to \mathsf{List}\ Carrier \to Carrier$$

We can now define the function fold which evaluates a big operator expression. It first applies a function $f : I \to Carrier$ to each element of an index list using the function map defined in the standard library:

$$\mathsf{map} : (I \to Carrier) \to \mathsf{List}\ I \to \mathsf{List}\ Carrier$$

fold then applies crush to combine them into a single value of the monoid's carrier type.[2]

[1]The name crush for this particular fold originated in Meertens (1996) and is combined in Gibbons and Oliveira (2009) with a monoidal constraint. In Haskell the corresponding function is called `mconcat`.

[2]Haskell's equivalent of our fold function is called `foldMap`.

```
fold : (I → Carrier) → List I → Carrier
fold f = crush ∘ map f
```

The following syntax declaration makes the connection between this function and big operators clearer:

```
syntax fold (λ x → e) v = Σ[ x ← v ] e
```

It has the effect of rewriting any expression of the form $\Sigma[\ x \leftarrow xs\ ]\ e$ into fold $(\lambda\ x \to e)$ xs. Note that the syntax keyword allows us to define new binding sites: the variable x is *bound* within the expression e. This effect cannot be achieved with mixfix operators.

The following two examples show how $\Sigma[\ i \leftarrow l\ ]\ f\ i$ evaluates. We first consider the case where $l = l_0 :: l_1 :: l_2 :: []$:

$$\begin{aligned}
\mathsf{fold}\ \mathsf{f}\ (l_0 :: l_1 :: l_2 :: []) &\equiv (\mathsf{crush} \circ \mathsf{map}\ \mathsf{f})\ (l_0 :: l_1 :: l_2 :: []) \\
&\equiv \mathsf{crush}\ (\mathsf{map}\ \mathsf{f}\ (l_0 :: l_1 :: l_2 :: [])) \\
&\equiv \mathsf{crush}\ (\mathsf{f}\ l_0 :: \mathsf{f}\ l_1 :: \mathsf{f}\ l_2 :: []) \\
&\equiv \mathsf{f}\ l_0 \bullet (\mathsf{f}\ l_1 \bullet (\mathsf{f}\ l_2)) \\
&\equiv \mathsf{f}\ l_0 \bullet \mathsf{f}\ l_1 \bullet \mathsf{f}\ l_2
\end{aligned}$$

In the last step, we are allowed to drop the parentheses because _•_ is associative by assumption. Next, we check what happens if l is empty, that is, $l = []$:

$$\begin{aligned}
\mathsf{fold}\ \mathsf{f}\ [] &\equiv (\mathsf{crush} \circ \mathsf{map}\ \mathsf{f})\ [] \\
&\equiv \mathsf{crush}\ (\mathsf{map}\ \mathsf{f}\ []) \\
&\equiv \mathsf{crush}\ [] \\
&\equiv \varepsilon
\end{aligned}$$

3.3 Intervals

Intervals of natural numbers are commonly used as indices in big operator expressions. This Section describes how intervals are defined for two types introduced in Section 2.1.1: natural numbers (ℕ) and Fin, which can be put into correspondence with some prefix of the natural numbers.

Defining intervals of type ℕ is straightforward. The module Bigop.Interval.Nat contains two functions:

```
upFrom : ℕ → ℕ → List ℕ
upFrom from zero       = []
upFrom from (suc len)  = from :: upFrom (suc from) len

range : ℕ → ℕ → List ℕ
range m n = upFrom m (n ∸ m)
```

We define two infix operators using range. "m …< n" evaluates to the interval of numbers from m up to but not including n. "m … n", on the other hand, does include n.

```
_…<_ = range

_…_ : ℕ → ℕ → List ℕ
m … n = range m (suc n)
```

upFrom m n evaluates to the interval containing n consecutive natural numbers, starting with m. range returns a list of natural numbers starting with m up to but not including n. In case $n \leq m$, range returns the empty list.

The infix operator _…<_ is just a synonym for range. The notation is meant to make it clear that the interval does not include the upper bound. On the other hand, _…_ explicitly includes the upper bound, so it defines a closed interval.

Bigop.Interval.Fin contains four definitions with the same names as the module presented above. Their types are:

```
      upFrom : (from len : ℕ) → List (Fin (from + len))
range _…<_ : ℕ → (to : ℕ) → List (Fin to)
          _…_ : ℕ → (to : ℕ) → List (Fin (suc to))
```

We omit the definitions of those functions here. They are more involved than the corresponding definitions presented above because we need to convert from ℕ to Fin and rewrite types to make the inductive definitions work.

Since Fin n is the type of natural numbers less than n, we can tell from the types of the functions that their upper bounds are as expected.

3.4 Filters

Sometimes it is useful to write the list of indices of a big operator expression as an interval out of which we only keep those indices which fulfil a certain property. The odd Gauss equation, for example, has as its right-hand side “the sum of all *odd* numbers from zero to $2n$”. In order to express such an equation in this framework, we need a way to filter out the even numbers. In this Section, we will define filters using a infix operator that combines well with the syntax for big operators presented in Section 3.2.3. Note that although the filter module was implemented as part of our big operators library, it does not depend on big operators or intervals, and could in the future be spun off into a separate library.

The filter operator _∥_ takes a list and a decidable predicate over the list’s element type, and returns the list containing only those elements of the input list which satisfy that predicate. Its definition is a straightforward inductive definition, with a case split on whether the predicate is satisfied: if yes, then the element is included in the result; otherwise it is dropped:

```
_∥_ : ∀ {a p} {A : Set a} {P : Pred A p} → List A → Decidable P → List A
[]         ∥ dec = []
(x :: xs)  ∥ dec with dec x
(x :: xs)  ∥ dec | yes _ = x :: (xs ∥ dec)
(x :: xs)  ∥ dec | no  _ = xs ∥ dec
```

Any decidable predicate can be converted into a function from the predicate's domain to Bool, mapping yes to true and no to false and dropping the evidence. This function is called ⌊_⌋ in the Agda standard library. ⌊_⌋ ∘ dec : A → Bool can then be passed to the function filter, also defined in the standard library, which filters a list using a function from the list's element type to Bool:

```
filter : ∀ {a} {A : Set a} → (A → Bool) → List A → List A
```

The following proof shows that the result of filtering a list using _∥_ is equal to calling filter with the converted decidable predicate as its first argument:

```
∥-filters : ∀ {a p} {A : Set a} {P : Pred A p} (xs : List A) (dec : Decidable P) →
            xs ∥ dec ≡ filter (⌊_⌋ ∘ dec) xs
∥-filters []         dec = refl
∥-filters (x :: xs)  dec with dec x
∥-filters (x :: xs)  dec | yes _ = cong (_::_ x) (∥-filters xs dec)
∥-filters (x :: xs)  dec | no  _ = ∥-filters xs dec
```

Bigop.Filter also defines the following important utility function ∁′, which takes a decidable predicate and returns its complement. For any x in the domain of predicate P, if x satisfies P, it returns evidence that x does not satisfy the complement of P, written ∁ P, and vice versa:

```
∁′ : ∀ {i p} {I : Set i} {P : Pred I p} → Decidable P → Decidable (∁ P)
∁′ p x with p x
∁′ p x | yes q  = no (λ ¬q → ¬q q)
∁′ p x | no ¬q  = yes (λ q → ¬q q)
```

In Bigop.Filter.Properties, we show a number of lemmas about filters and decidable properties defined in this way. In each of them, we pick an element from the list and examine whether it satisfies the predicate. They can be considered variants of list induction. In some proofs it is more convenient to perform induction on the list of indices from the head (the first element of the list). This can be achieved simply by pattern matching on a non-empty list using the constructor _::_. In other proofs we may want to start with the last element. xs ::ʳ x (called *snoc* in Lisp) abbreviates xs ++ x :: [], and allows us to split a list into its last element (x) and everything preceding it (xs).

Omitting the parameters $\{i\ \ell : \mathsf{Level}\}\ \{I : \mathsf{Set}\ i\}\ \{P : \mathsf{Pred}\ I\ \ell\}\ x\ xs\ (p : \mathsf{Decidable}\ P)$, the types of the filter lemmas are as follows:

$$\begin{aligned}
\mathsf{head{-}yes} &: P\,x \to (x :: xs) \parallel p \equiv x :: (xs \parallel p)\\
\mathsf{last{-}yes} &: P\,x \to (xs ::^{r} x) \parallel p \equiv (xs \parallel p) ::^{r} x\\
\mathsf{head{-}no} &: \neg\, P\,x \to (x :: xs) \parallel p \equiv xs \parallel p\\
\mathsf{last{-}no} &: \neg\, P\,x \to (xs ::^{r} x) \parallel p \equiv xs \parallel p\\
\mathsf{head{-}\complement{-}yes} &: P\,x \to (x :: xs) \parallel \complement'\,p \equiv xs \parallel \complement'\,p\\
\mathsf{head{-}\complement{-}no} &: \neg\, P\,x \to (x :: xs) \parallel \complement'\,p \equiv x :: (xs \parallel \complement'\,p)
\end{aligned}$$

All six of the above are proved using the following more general lemmas (all of which take the implicit parameters $\{i\ \ell : \mathsf{Level}\}\ \{I : \mathsf{Set}\ i\}\ \{P : \mathsf{Pred}\ I\ \ell\}$):

$$\begin{aligned}
\mathsf{\parallel{-}distrib} &: \forall\, xs\ ys\ (p : \mathsf{Decidable}\ P) \to (xs \mathbin{+\!+} ys) \parallel p \equiv (xs \parallel p) \mathbin{+\!+} (ys \parallel p)\\
\mathsf{\parallel{-}step{-}yes} &: \forall\, x\ (p : \mathsf{Decidable}\ P) \to P\,x \to [\,x\,] \parallel p \equiv [\,x\,]\\
\mathsf{\parallel{-}step{-}no} &: \forall\, x\ (p : \mathsf{Decidable}\ P) \to \neg\, P\,x \to [\,x\,] \parallel p \equiv []
\end{aligned}$$

In addition, we prove $\mathsf{ordinals{-}filter}$ (see its type below). It states that the interval of length n starting at m contains the number k exactly once, provided that $m \leq k$ and $k < m + n$. “Containing k exactly once” is expressed in terms of filtering by the decidable predicate $\overset{?}{=}\mathsf{N}\ k$, which holds only for natural numbers equal to k. If the result of applying this filter to an interval yields a singleton list containing only k, then k must have appeared exactly once in the interval.

$$\mathsf{ordinals{-}filter} : \forall\, m\ n\ k \to m \leq k \to (k{<}m{+}n : k < m + n) \to \mathsf{upFrom}\ m\ n \parallel (\overset{?}{=}\mathsf{N}\ k) \equiv k :: []$$

This lemma is used in the proof that the identity matrix really is the identity element of the semiring of square matrices (see Section 4.4).

3.5 Properties of big operators

In this Section, we discuss how the properties of the underlying operators $\_{+}\_$ and $\_{*}\_$ determine the properties of their induced big operators.

The lemmas in this section reside in Bigop.Properties. They are intended to be used as follows: assuming we have some commutative monoid CM : CommutativeMonoid. Then the following line will bring the lemmas concerning commutative monoids into scope under the module name Σ:

$$\mathsf{module}\ \Sigma = \mathsf{Bigop.Properties.CommutativeMonoid}\ CM$$

For convenience, Bigop.Properties.CommutativeMonoid re-exports all lemmas about monoids and Bigop.Properties.SemiringWithoutOne re-exports the commutative monoid lemmas.

3.5.1 Monoid lemmas

Monoids are endowed with an identity and an associativity law. Based on these two properties, there are a few things we can say about what happens when a big operator is defined on a monoid.

Lifted identity. The identity law can be lifted to give the following equivalence for the monoid's big operator:

$$(\text{identity}) \qquad \Sigma[\, i \leftarrow is \,]\; \varepsilon \approx \varepsilon$$

Distributivity over _++_. It follows from the monoid associativity law that big operators distribute over the list append function _++_ as follows:

$$(\text{join}) \qquad \Sigma[\, i \leftarrow xs \,]\; f\; i + \Sigma[\, i \leftarrow ys \,]\; f\; i \approx \Sigma[\, i \leftarrow xs \mathbin{++} ys \,]\; f\; i$$

Congruence. Given $xs \equiv ys$ and $\forall\, x \rightarrow f\; x \approx g\; x$, we can show that

$$(\text{cong}) \qquad \Sigma[\, x \leftarrow xs \,]\; f\; x \approx \Sigma[\, y \leftarrow ys \,]\; f\; y$$

3.5.2 Commutative monoid lemmas

If the binary operation _+_ is commutative as well as associative, the big operator defined on it has more properties that we can use in proofs.

Distributivity over _+_. Combining associativity and distributivity of _+_, we can show that big operators distribute as follows:

$$(\text{merge}) \qquad \Sigma[\, x \leftarrow xs \,]\; f\; x + \Sigma[\, x \leftarrow xs \,]\; g\; x \approx \Sigma[\, x \leftarrow xs \,]\; f\; x + g\; x$$

Swapping big operators. If the underlying operator _+_ is commutative, the order in which big operators are evaluated does not matter:

$$(\text{swap}) \qquad \Sigma[\, x \leftarrow xs \,]\; (\Sigma[\, y \leftarrow ys \,]\; f\; x\; y) \approx \Sigma[\, y \leftarrow ys \,]\; (\Sigma[\, x \leftarrow xs \,]\; f\; x\; y)$$

Splitting by a predicate. Any list can be split into two lists, one containing those elements which satisfy a decidable predicate ($xs \parallel dec$) and one containing those which do not ($xs \parallel \mathbb{C}'\; dec$). Assuming commutativity, we can split the index list of a big operator expression using any decidable predicate and add them together to get the same result as taking the sum over the original list:

$$(\text{split--P}) \qquad \Sigma[\, x \leftarrow xs \,]\; f\; x \approx \Sigma[\, x \leftarrow xs \parallel dec \,]\; f\; x + \Sigma[\, x \leftarrow xs \parallel \mathbb{C}'\; dec \,]\; f\; x$$

3.5.3 "Semiring without one" lemmas

A "semiring without one" consists of a commutative monoid over an operation _+_ and a semigroup over an operation _∗_ with a zero element (annihilator). Additionally, _∗_ distributes over _+_. This allows us to prove two distributivity laws for constants:

$$
\begin{array}{ll}
(\mathsf{distr}^{\mathsf{l}}) & a * (\Sigma[\, x \leftarrow xs \,]\; f\; x) \approx \Sigma[\, x \leftarrow xs \,]\; a * f\; x \\
(\mathsf{distr}^{\mathsf{r}}) & (\Sigma[\, x \leftarrow xs \,]\; f\; x) * a \approx \Sigma[\, x \leftarrow xs \,]\; f\; x * a
\end{array}
$$

3.5.4 Boolean algebra lemmas

The lemmas about big operators on monoids, commutative monoids and semirings presented above were directly relevant to the theorems we aimed to prove (see Chapter 4 as well as Appendix B and Appendix C). To demonstrate that our approach scales to more complex algebraic structures, we proved the big operator version of the de Morgan laws for arbitrary Boolean algebras. Two examples of Boolean algebras are: Booleans with the two operators logical *and* and logical *or*, and sets with intersection and union.

In order to make the propositions and proofs more readable, we added two syntax definitions as synonyms for fold (see Section 3.2):

```
syntax fold (λ x → e) v = ⋁[ x ← v ] e
syntax fold (λ x → e) v = ⋀[ x ← v ] e
```

Using this notation, we proved that for any Boolean algebra the following variants of the de Morgan laws hold:

$$
\begin{array}{ll}
(\mathsf{deMorgan}_1) & \neg\, (\bigwedge[\, x \leftarrow xs \,]\; f\; x) \approx \bigvee[\, x \leftarrow xs \,]\; \neg\, f\; x \\
(\mathsf{deMorgan}_2) & \neg\, (\bigvee[\, x \leftarrow xs \,]\; f\; x) \approx \bigwedge[\, x \leftarrow xs \,]\; \neg\, f\; x
\end{array}
$$

Boolean algebras provide a rich structure, and more interesting properties of this structure could be lifted into lemmas about their big operators. The point here is just to demonstrate that the framework we developed is very general, for example allowing us to write proofs concerning the interaction between two big operators defined on the same algebraic structure.

Chapter 4

Square matrices over semirings

In this Chapter we present a proof that square matrices over a semiring themselves form a semiring. One explicit success criterion of this project was to make it possible to write this proof using our library.

Section 4.2 introduces various definitions. In Section 4.3 we show that square matrices and matrix addition constitute a commutative monoid with an annihilator. Section 4.4 proves that square matrices and matrix multiplication form a monoid. In Section 4.5 we show that matrix multiplication distributes over matrix addition.

4.1 Matrices

In order to prove that square matrices over semirings are again semirings, it was necessary to formalise matrices first (the Agda standard library currently lacks a matrix library). This module is independent from the rest of the project.

A matrix is defined as a vector of row vectors over some carrier type. Matrix A r c is the type of $r \times c$ matrices over carrier A.

```
Matrix : ∀ {a} (A : Set a) → ℕ → ℕ → Set a
Matrix A r c = Vec (Vec A c) r
```

lookup i j m returns the jth element of row i of the matrix m, as expected. The second function allows us to write A [i , j] instead.

```
lookup : ∀ {r c a} {A : Set a} → Fin r → Fin c → Matrix A r c → A
lookup i j m = V.lookup j (V.lookup i m)

_[_,_] : ∀ {r c a} {A : Set a} → Matrix A r c → Fin r → Fin c → A
m [ i , j ] = lookup i j m
```

tabulate populates a matrix using a function that takes the row and column index to an element of the matrix by applying that function to each position in the matrix. As an example, the transposition function is included too. Note how the change in the shape of the matrix is reflected in the return type of the function, where the number of rows and columns are swapped.

```
tabulate : ∀ {r c a} {A : Set a} → (Fin r → Fin c → A) → Matrix A r c
tabulate f = V.tabulate (λ r → V.tabulate (λ c → f r c))

transpose : ∀ {r c a} {A : Set a} → Matrix A r c → Matrix A c r
transpose m = tabulate (λ c r → lookup r c m)
```

Lastly, the lemma lookup∘tabulate shows that if a matrix was created by tabulating a function f, then looking up the element at row i and column j returns $f\ i\ j$. The proof uses a similar lemma for vectors, VP.lookup∘tabulate.

```
lookup∘tabulate : ∀ {a n} {A : Set a} {f : Fin n → Fin n → A} i j →
                  lookup i j (tabulate f) ≡ f i j
lookup∘tabulate {f = f} i j = begin
  V.lookup j (V.lookup i (V.tabulate (V.tabulate ∘ f)))
    ≡⟨ P.cong (V.lookup j) (VP.lookup∘tabulate (V.tabulate ∘ f) i) ⟩
  V.lookup j (V.tabulate (f i))
    ≡⟨ VP.lookup∘tabulate (f i) j ⟩
  f i j ∎
```

Next we define a relation between matrices based on a relation between their elements:

```
Pointwise : ∀ {s t ℓ} {S : Set s} {T : Set t} (_~_ : REL S T ℓ)
            {m n} → Matrix S m n → Matrix T m n → Set ℓ
Pointwise _~_ A B = ∀ r c → A [ r , c ] ~ B [ r , c ]
```

The intuition for this definition is this: in order to show that the Pointwise _~_ relation holds between two matrices, we must give a function which, for any row index r and any column index c, produces evidence that the relation _~_ holds between the elements of the two matrices at this point.

We would like to use Pointwise _~_ as an equivalence relation in proofs. PW−isEquivalence shows that if _ _ is an equivalence relation, then so is its pointwise lifting:

```
PW−isEquivalence : ∀ {a ℓ} {A : Set a} {_≈_ : Rel A ℓ} {m n} →
                   IsEquivalence _≈_ → IsEquivalence (Pointwise _≈_ {m = m} {n})
PW−isEquivalence {_≈_ = ≈} eq = record
  { refl   = λ r c → refl
  ; sym    = λ eq r c → sym (eq r c)
  ; trans  = λ eq₁ eq₂ r c → trans (eq₁ r c) (eq₂ r c) }
  where open IsEquivalence eq
```

Let us consider sym in more detail. The property Symmetric ≈ is defined as $\forall \{A\ B\} \to A \approx B \to B \approx A$. That is, it transforms evidence of $eq : A \approx B$ into evidence that $B \approx A$, which is just a synonym for $\forall\ r\ c \to B\,[\,r\,,c\,] \approx A\,[\,r\,,c\,]$. In order to construct a function of this type, we abstract over r and c and then apply the symmetry law of the underlying equivalence sym to $eq\ r\ c$ like so:

$$\begin{aligned}
\lambda\, r\, c \to eq\ r\ c &: A \approx B \\
&: \forall\, r\, c \to A\,[\, r\, ,c\,] \approx B\,[\, r\, ,c\,] \\
\lambda\, r\, c \to \mathsf{sym}\ (eq\ r\ c) &: \forall\, r\, c \to B\,[\, r\, ,c\,] \approx A\,[\, r\, ,c\,] \\
&: B \approx A
\end{aligned}$$

Our symmetry proof for _≈_ is thus

$$\approx\!-\mathsf{sym}\ eq = \lambda\, r\, c \to \mathsf{sym}\ (eq\ r\ c)$$

Reflexivity and transitivity are proved in a similar fashion.

4.2 Definitions

In this Section, we define matrix addition and multiplication, the zero matrix and the identity matrix. All the code in this Chapter resides in a module that is parameterised over the underlying semiring and the size n of the square matrices. The following declaration has the effect of fixing the variables n, c, ℓ and *semiring* in the module's body:

```
module SemiringProof (n : ℕ) {c ℓ} (semiring : Semiring c ℓ) where
```

In the next listing, we bring the underlying semiring with its carrier type (*Carrier*), special elements (0#, 1#), operators (_+_, _∗_) and substructures into scope:[1] the commutative monoid, monoid and semigroup over _+_ (+−*commutativeMonoid*, +−*monoid*, +−*semigroup*); the monoid and semigroup over _∗_ (∗−*monoid*, ∗−*semigroup*); and the “semiring without one” (*semiringWithoutOne*, a semiring-like structure without an identity for _∗_).

Next, the equivalence relation _≡_ of the underlying setoid on Carrier and its reflexive, symmetric and transitive laws (refl, sym, trans) are brought into scope. We make the sum syntax from the Bigop.Core.Fold module available and open the modules containing lemmas about ordinals, equational reasoning functionality in the element setoid (EqReasoning) and the module for equational reasoning with propositional equality (≡−Reasoning). In order to avoid name clashes, the functions begin_, _≡⟨_⟩_ and _∎ are renamed to start_, _≡⟨_⟩_ and _□, respectively.

We define M as a shorthand for the type of square matrices of size n over the carrier of the underlying semiring. The pointwise lifting of the equivalence relation between elements is named _≈_. Matrix and Pointwise are defined in Section 4.1.

[1] Here *substructures* refers to the weaker structures that are automatically derived from the given algebraic structure by subtracting properties, operators or special elements. For example, any commutative monoid gives rise to a monoid if we take away the commutative law.

```
M : Set c
M = Matrix Carrier n n

_≋_ : Rel M ℓ
_≋_ = Pointwise _≈_

≋-isEquivalence : IsEquivalence _≋_
≋-isEquivalence = PW-isEquivalence isEquivalence
```

Next we define matrix addition _⊕_ and multiplication _⊗_. Addition works pointwise. The definition of tabulate is given in Section 4.1.

```
_⊕_ : M → M → M
A ⊕ B = tabulate (λ r c → A [ r , c ] + B [ r , c ])
```

Using Σ-syntax, multiplication can be defined in a concise way:

```
mult : M → M → Fin n → Fin n → Carrier
mult A B r c = Σ[ i ← 0 …< n ] A [ r , i ] * B [ i , c ]

_⊗_ : M → M → M
A ⊗ B = tabulate (mult A B)
```

Note how the definition of mult resembles the component-wise definition of matrix multiplication in standard mathematical notation:

$$(A\,B)_{r,c} = \sum_{i \leftarrow 0 \ldots < n} A_{r,i}\, B_{i,c}$$

The matrix 0M is the identity for matrix addition and the annihilator for matrix multiplication. All of its elements are set to the zero element of the underlying semiring.

```
0M : M
0M = tabulate (λ r c → 0#)
```

1M is the identity for matrix multiplication. Its definition relies on the function diag, which returns 1# (the multiplicative identity of the underlying semiring) if its arguments are equal and 0# (the additive identity and multiplicative annihilator of the underlying semiring) if they are different.

```
diag : {n : ℕ} → Fin n → Fin n → Carrier
diag  zeroF     zeroF     =  1#        – r ≡ c
diag  zeroF     (sucF c)  =  0#        – r ≢ c
diag  (sucF r)  zeroF     =  0#        – r ≢ c
diag  (sucF r)  (sucF c)  =  diag r c  – recursive case
```

The identity matrix 1M is defined as the result of tabulating diag:

```
1M : M
1M = tabulate diag
```

Note that there are many ways to define the function diag. One alternative would be to use an explicit equality test. The inductive definition given above turned out to be easiest to work with in proofs.

4.3 Properties of matrix addition

In this Section, we show that square matrices and matrix addition form a commutative monoid.

Congruence

Here we prove that matrix addition preserves equivalence, that is, $A \approxeq A' \to B \approxeq B' \to A \oplus B \approxeq A' \oplus B'$.

$$\begin{aligned}(A \oplus B)_{r,c} &\approx A_{r,c} + B_{r,c} && (4.1)\\ &\approx A'_{r,c} + B'_{r,c} && (4.2)\\ &\approx (A' \oplus B')_{r,c} && (4.3)\end{aligned}$$

To show that two matrices are $\approxeq$-equivalent, we need to prove that their elements are $\approx$-equivalent. This dictates the structure of ⊕−cong and all other proofs of matrix equivalence: an (equational reasoning style) proof of element-wise equivalence, abstracted over the row and column index.

In the inner proof, we first expand the definitions of _⊕_ and _⊗_, then use the properties of the underlying operators _+_ and _∗_ (in this case, congruence of _+_), and finally re-assemble the matrix operators.

```
⊕−cong : ∀ {A A′ B B′} → A ≋ A′ → B ≋ B′ → A ⊕ B ≋ A′ ⊕ B′
⊕−cong {A} {A′} {B} {B′} eq₀ eq₁ = λ r c →
  begin
{- 4.1 -} (A ⊕ B) [ r , c ]        ≡⟨ lookup∘tabulate r c ⟩
{- 4.2 -} A [ r , c ]  + B [ r , c ] ≈⟨ eq₀ r c ⟨ +−cong ⟩ eq₁ r c ⟩
{- 4.3 -} A′ [ r , c ] + B′ [ r , c ] ≡⟨ P.sym (lookup∘tabulate r c) ⟩
          (A′ ⊕ B′) [ r , c ]
  ∎
```

Since the only law used in this proof is +−cong, the semigroup over _+_ induced by the underlying semiring is sufficient to prove that matrix addition is congruent.

Associativity

The next proof shows that matrix addition is associative, that is, $(A \oplus B) \oplus C \approxeq A \oplus (B \oplus C)$. Since matrix addition is defined as elementwise addition, the proof of elementwise equivalence has the exact same structure as the congruence proof above: unfold the definition of _⊕_; use the appropriate properties (associativity in this case) of the elementwise addition _+_; fold back into matrix addition. In standard mathematical notation, associativity of matrix addition can be proved as follows:

$$((A \oplus B) \oplus C)_{r,c} \approx (A_{r,c} + B_{r,c}) + C_{r,c} \tag{4.4}$$
$$\approx A_{r,c} + (B_{r,c} + C_{r,c}) \tag{4.5}$$
$$\approx (A \oplus (B \oplus C))_{r,c} \tag{4.6}$$

The auxiliary functions ⟨⊕⟩⊕−expand and ⊕⟨⊕⟩−expand simply unfold the nested matrix additions.

```
  ⊕−assoc : ∀ A B C → (A ⊕ B) ⊕ C ≈ A ⊕ (B ⊕ C)
  ⊕−assoc A B C = λ r c →
     begin
{- 4.4 -}  ((A ⊕ B) ⊕ C) [ r , c ]           ≡⟨ ⟨⊕⟩⊕−expand r c ⟩
{- 4.5 -}  (A [ r , c ] +  B [ r , c ]) + C [ r , c ]  ≈⟨ +−assoc _ _ _ ⟩
{- 4.6 -}  A [ r , c ] + (B [ r , c ]  + C [ r , c ])  ≡⟨ P.sym (⊕⟨⊕⟩−expand r c) ⟩
           (A ⊕ (B ⊕ C)) [ r , c ]
     ∎
```

Again, a semigroup over _+_ provides sufficient structure to allow the proof to go through.

Left identity

In order to prove that the zero matrix is an identity for _⊕_, we first expand the definition of matrix addition. Then by definition, 0M [r , c] ($\mathbf{0}_{r,c}$ in mathematical notation) is equal to 0# for any r and c. The left identity law of the underlying monoid over _+_ then justifies the equivalence:

$$(\mathbf{0} \oplus A)_{r,c} \approx \mathbf{0}_{r,c} + A_{r,c} \tag{4.7}$$
$$\approx 0 + A_{r,c} \tag{4.8}$$
$$\approx A_{r,c} \tag{4.9}$$

```
  ⊕−identityˡ : ∀ A → 0M ⊕ A ≈ A
  ⊕−identityˡ A = λ r c →
     begin
{- 4.7 -}  (0M ⊕ A) [ r , c ]         ≡⟨ lookup∘tabulate r c ⟩
{- 4.8 -}  0M [ r , c ] +  A [ r , c ] ≡⟨ P.cong₂ _+_ (lookup∘tabulate r c) P.refl ⟩
{- 4.9 -}  0# +            A [ r , c ] ≈⟨ proj₁ +−identity _ ⟩
                           A [ r , c ] ∎
```

Note that this proof makes use of +−identity, which +−semigroup does not provide. This is why we open the monoid over _+_ in the identity proof above.

Commutativity of matrix addition

The commutativity proof follows the now-familiar pattern: we use the definition of matrix addition, apply the commutativity law of elementwise addition and finally we rewrite the term again using the definition of addition. Again, we present the proof in standard mathematical notation and then in Agda:

$$
\begin{aligned}
(A \oplus B)_{r,c} &\approx A_{r,c} + B_{r,c} && (4.10)\\
&\approx B_{r,c} + A_{r,c} && (4.11)\\
&\approx (B \oplus A)_{r,c} && (4.12)
\end{aligned}
$$

```
⊕−comm : ∀ A B → A ⊕ B ≈ B ⊕ A
⊕−comm A B = λ r c →
  begin
{- 4.10 -} (A ⊕ B) [ r , c ]         ≡⟨ lookup∘tabulate r c ⟩
{- 4.11 -}  A [ r , c ] + B [ r , c ] ≈⟨ +−comm _ _ ⟩
{- 4.12 -}  B [ r , c ] + A [ r , c ] ≡⟨ P.sym (lookup∘tabulate r c) ⟩
           (B ⊕ A) [ r , c ]
  ∎
```

Here _+_ must be a commutative operator for the proof to go through.

Matrix addition: a commutative monoid

Putting all the lemmas in this Section together, we have shown that matrix addition forms a commutative monoid over square matrices with 0M as its identity:

```
⊕−isCommutativeMonoid : IsCommutativeMonoid _≈_ _⊕_ 0M
⊕−isCommutativeMonoid = record
  { isSemigroup = record
    { isEquivalence = ≈−isEquivalence
    ; assoc       = ⊕−assoc
    ; •−cong      = ⊕−cong
    }
  ; identityˡ     = ⊕−identityˡ
  ; comm          = ⊕−comm
  }
```

4.4 Properties of matrix multiplication

In this Section we prove that matrix multiplication is monoidal. Additionally, a proof is given that 0M is a left zero for matrix multiplication.

Congruence of matrix multiplication

In this proof we need to use both Σ.cong and ∗−cong to replace equals by equals in a multiplication wrapped in a sum. The structure of the proof is unchanged from the last Section.

See Section 3.5.1 for a description of the lemmas contained in Bigop.Properties.Monoid.

$$\begin{aligned}(A \otimes B)_{r,c} &\approx \sum_{i \leftarrow 0 \ldots < n} A_{r,i}\, B_{i,c} && (4.13)\\ &\approx \sum_{i \leftarrow 0 \ldots < n} A'_{r,i}\, B'_{i,c} && (4.14)\\ &\approx (A' \otimes B')_{r,c} && (4.15)\end{aligned}$$

```
⊗−cong : ∀ {A A′ B B′} → A ≈ A′ → B ≈ B′ → A ⊗ B ≈ A′ ⊗ B′
⊗−cong {A} {A′} {B} {B′} eq₁ eq₂ = λ r c →
  begin
    (A ⊗ B) [ r , c ]
{- 4.13 -}  ≡⟨ lookup∘tabulate r c ⟩
    Σ[ i ← 0 …< n ] A [ r , i ] ∗ B [ i , c ]
{- 4.14 -}  ≈⟨ Σ.cong (0 …< n) P.refl (λ i → ∗−cong (eq₁ r i) (eq₂ i c)) ⟩
    Σ[ i ← 0 …< n ] A′ [ r , i ] ∗ B′ [ i , c ]
{- 4.15 -}  ≡⟨ P.sym (lookup∘tabulate r c) ⟩
    (A′ ⊗ B′) [ r , c ]
  ∎
```

We can read off the open statements that this proof requires a semigroup over _∗_ and a monoid over _+_.

Left zero for matrix multiplication

In this proof that 0M is the left zero for _⊗_, that is, 0M ⊗ A ≈ 0M, we use two lemmas from Bigop.Properties.

$$\begin{aligned}(\mathbf{0} \otimes A)_{r,c} &\approx \sum_{i \leftarrow 0 \ldots < n} \mathbf{0}_{r,i}\, A_{i,c} && (4.16)\\ &\approx \sum_{i \leftarrow 0 \ldots < n} 0\, A_{i,c} && (4.17)\\ &\approx 0 \cdot \sum_{i \leftarrow 0 \ldots < n} A_{i,c} && (4.18)\\ &\approx 0 && (4.19)\\ &\approx \mathbf{0}_{r,c} && (4.20)\end{aligned}$$

```
  M-zero' : ∀ A → 0M ⊗ A ≈ 0M
  M-zero' A = λ r c →
    begin
{- 4.16 -} (0M ⊗ A) [ r , c ] ≡⟨ lookup∘tabulate r c ⟩
           Σ[ i ← 0 …< n ] 0M [ r , i ] * A [ i , c ]
             ≈⟨ Σ.cong  (0 …< n) P.refl
                         (λ i → reflexive (lookup∘tabulate r i) ⟨ *-cong ⟩ refl) ⟩
           Σ[ i ← 0 …< n ] 0# * A [ i , c ]
             ≈⟨ Σ.cong (0 …< n) P.refl (λ i → proj₁ zero _) ⟩
           Σ[ i ← 0 …< n ] 0#
             ≈⟨ Σ.identity (0 …< n) ⟩
{- 4.20 -} 0#                       ≡⟨ P.sym (lookup∘tabulate r c) ⟩
           0M [ r , c ]
    ∎
```

Let us consider the second step of the proof in detail. The aim is to use Σ.cong to show

$$\Sigma[\, i \leftarrow 0 \ldots\!< n \,]\ 0M\,[\, r , i \,] * A\,[\, i , c \,] \approx \Sigma[\, i \leftarrow 0 \ldots\!< n \,]\ 0\# * A\,[\, i , c \,]$$

On both sides of the equivalence, we take the sum over (0 …< n). The proof that the lists are propositionally equal is therefore just P.refl.

We now need to prove that 0M [r , i] * A [i , c] ≈ 0# * A [i , c] for all i. The outer form of this expression is a multiplication. Since our goal is to replace equal subterms by equal subterms, we need to use the appropriate congruence rule, *-cong. The right hand sides of the two multiplications are both A [i , c], so they are trivially equivalent by refl.

0M [r , i] is propositionally equal to 0# by (lookup∘tabulate r i). Equivalence in _≈_ follows from propositional equality by reflexivity, which proves that the left hand sides of the multiplications are ≈-equivalent too.

Putting this all together, we have built a term

```
Σ.cong (0 …< n) P.refl (λ i → reflexive (lookup∘tabulate r i) ⟨ *-cong ⟩ refl) ⟩
```

proving the statement above.

In the remainder of the proof, we first apply the zero law of the underlying semiring and then Σ.identity, which shows that the sum of any number of zeros is zero. Right-distributivity is proved in a similar way. The proof is omitted here, but it is included in the Agda source files of the project.

Associativity of matrix multiplication

This proof is more involved than the associativity proof for matrix addition. The argument runs as follows:

$$\begin{aligned}
((A \otimes B) \otimes C)_{r,c} &\approx \sum_{i \leftarrow 0 \ldots < n} \left(\sum_{j \leftarrow 0 \ldots < n} A_{r,j} B_{j,i} \right) C_{i,c} && \text{by the definition of } \otimes && (4.21)\\
&\approx \sum_{i \leftarrow 0 \ldots < n} \sum_{j \leftarrow 0 \ldots < n} (A_{r,j} B_{j,i}) C_{i,c} && \text{by right-distributivity} && (4.22)\\
&\approx \sum_{j \leftarrow 0 \ldots < n} \sum_{i \leftarrow 0 \ldots < n} (A_{r,j} B_{j,i}) C_{i,c} && \text{swapping the outer sums} && (4.23)\\
&\approx \sum_{j \leftarrow 0 \ldots < n} \sum_{i \leftarrow 0 \ldots < n} A_{r,j} (B_{j,i} C_{i,c}) && \text{by associativity} && (4.24)\\
&\approx \sum_{j \leftarrow 0 \ldots < n} A_{r,j} \sum_{i \leftarrow 0 \ldots < n} B_{j,i} C_{i,c} && \text{by left-distributivity} && (4.25)\\
&\approx (A \otimes (B \otimes C))_{r,c} && \text{by the definition of } \otimes && (4.26)
\end{aligned}$$

In the Agda proof, we use the appropriate congruence rules to replace subterms by equivalent subterms. Steps (3.4) and (3.5) have been factored into a separate function inner to make the proof more readable. ⟨⊗⟩⊗−expand and ⊗⟨⊗⟩−expand unfold the nested matrix multiplications.

```
⊗−assoc : ∀ A B C → (A ⊗ B) ⊗ C ≋ A ⊗ (B ⊗ C)
⊗−assoc A B C = λ r c →
  begin
    ((A ⊗ B) ⊗ C) [ r , c ]
{- 4.21 -} ≈⟨ ⟨⊗⟩⊗−expand r c ⟩
    Σ[ i ← 0 ...< n ] (Σ[ j ← 0 ...< n ] A [ r , j ] ∗ B [ j , i ]) ∗ C [ i , c ]
{- 4.22 -} ≈⟨ Σ.cong (0 ...< n) P.refl (λ i → Σ.distrʳ _ _ (0 ...< n)) ⟩
    Σ[ i ← 0 ...< n ] Σ[ j ← 0 ...< n ] (A [ r , j ] ∗ B [ j , i ]) ∗ C [ i , c ]
{- 4.23 -} ≈⟨ Σ.swap _ (0 ...< n) (0 ...< n) ⟩
    Σ[ j ← 0 ...< n ] Σ[ i ← 0 ...< n ] (A [ r , j ] ∗ B [ j , i ]) ∗ C [ i , c ]
      ≈⟨ Σ.cong (0 ...< n) P.refl (inner r c) ⟩
    Σ[ j ← 0 ...< n ] A [ r , j ] ∗ (Σ[ i ← 0 ...< n ] B [ j , i ] ∗ C [ i , c ])
{- 4.26 -} ≈⟨ sym $ ⊗⟨⊗⟩−expand r c ⟩
    (A ⊗ (B ⊗ C)) [ r , c ]
  ∎
```

Within ⊗−assoc, inner is used which proves steps (4.24) and (4.23):

```
inner : ∀ r c j → Σ[ i ← 0 …< n ] (A [ r , j ] * B [ j , i ]) * C [ i , c ] ≈
                    A [ r , j ] * (Σ[ i ← 0 …< n ] B [ j , i ] * C [ i , c ])
inner r c j =
  begin
    Σ[ i ← 0 …< n ] (A [ r , j ] * B [ j , i ]) * C [ i , c ]
{- 4.24 -} ≈⟨ Σ.cong (0 …< n) P.refl (λ i → *−assoc _ _ _) ⟩
    Σ[ i ← 0 …< n ] A [ r , j ] * (B [ j , i ] * C [ i , c ])
{- 4.25 -} ≈⟨ sym (Σ.distrˡ _ _ (0 …< n)) ⟩
    A [ r , j ] * (Σ[ i ← 0 …< n ] B [ j , i ] * C [ i , c ])
  ∎
```

Left identity for matrix multiplication

This is the longest of the semiring proofs. We show that 1M ⊗ $A \approx A$ for all A. The key idea here is that for any term involving 1M, we can perform a case split on whether the row r and column c are equal. If they are, then 1M [r , c] ≡ 1# by 1M−diag. If not, then by 1M−∁−diag we have 1M [r , c] ≡ 0#:

The justification for the identity law in mathematical notation is as follows:

$$
\begin{aligned}
(\mathbf{1} \otimes A)_{r,c} &\approx \sum_{i \leftarrow 0 \ldots < n} \mathbf{1}_{r,i}\, A_{i,c} && (4.27)\\
&\approx \left(\sum_{\substack{i \leftarrow 0 \ldots < n \\ r \equiv i}} \mathbf{1}_{r,i}\, A_{i,c} \right) + \left(\sum_{\substack{i \leftarrow 0 \ldots < n \\ r \not\equiv i}} \mathbf{1}_{r,i}\, A_{i,c} \right) && (4.28)\\
&\approx \left(\sum_{\substack{i \leftarrow 0 \ldots < n \\ r \equiv i}} 1 \cdot A_{i,c} \right) + \left(\sum_{\substack{i \leftarrow 0 \ldots < n \\ r \not\equiv i}} 0 \cdot A_{i,c} \right) && (4.29)\\
&\approx \left(\sum_{\substack{i \leftarrow 0 \ldots < n \\ r \equiv i}} A_{i,c} \right) + 0 && (4.30)\\
&\approx \sum_{\substack{i \leftarrow 0 \ldots < n \\ r \equiv i}} A_{i,c} && (4.31)\\
&\approx A_{r,c} && (4.32)
\end{aligned}
$$

After unfolding the definition of ⊗, we split the list (0 …< n) that is being summed over by the decidable predicate ($\overset{?}{=}$ r) using Σ.split−P. This lets us consider the cases $r \equiv i$ and $r \not\equiv i$ separately. The function ≡−step deals with the first case. From $r \equiv i$ follows 1M [r , i] ≡ 1#. Using distributivity and the identity law for _*_, we can deduce that

$$\Sigma[\, i \leftarrow 0 \ldots < n \parallel \overset{?}{=} r \,]\ 1M\ [\, r , i \,] * A\ [\, i , c \,] \approx \Sigma[\, i \leftarrow 0 \ldots < n \parallel \overset{?}{=} r \,]\ A\ [\, i , c \,]$$

```
≡−step : ∀ (A : M) r c → Σ[ i ← 0 …< n ∥ ≟ r ] 1M [ r , i ] * A [ i , c ] ≈
                          Σ[ i ← 0 …< n ∥ ≟ r ] A [ i , c ]
≡−step A r c =
  begin
```

```
      Σ[ i ← 0 …< n ∥ ≟ r ] 1M [ r , i ] ∗ A [ i , c ]
        ≈⟨ Σ.cong-P (0 …< n) (≟ r)
                      (λ i r≡i → reflexive (1M-diag r≡i) ⟨ ∗-cong ⟩ refl) ⟩
      Σ[ i ← 0 …< n ∥ ≟ r ] 1# ∗ A [ i , c ]
        ≈⟨ sym $ Σ.distrˡ _ 1# (0 …< n ∥ ≟ r) ⟩
      1# ∗ (Σ[ i ← 0 …< n ∥ ≟ r ] A [ i , c ])
        ≈⟨ proj₁ ∗-identity _ ⟩
      Σ[ i ← 0 …< n ∥ ≟ r ] A [ i , c ]
    ∎
```

Otherwise, ≢-step assumes that $r \not\equiv i$. It follows that 1M [r , i] ≡ 0#. By distributivity and the zero law of the underlying semiring we then have

$$\Sigma[\, i \leftarrow 0 \ldots\!< n \parallel \complement' (\overset{?}{=} r) \,]\ 1M\,[\, r , i \,] * A\,[\, i , c \,] \approx 0\#$$

```
  ≢-step : ∀ (A : M) r c → Σ[ i ← 0 …< n ∥ ∁′ (≟ r) ] 1M [ r , i ] ∗ A [ i , c ] ≈ 0#
  ≢-step A r c =
    begin
      Σ[ i ← 0 …< n ∥ ∁′ (≟ r) ] 1M [ r , i ] ∗ A [ i , c ]
        ≈⟨ Σ.cong-P (0 …< n) (∁′ (≟ r))
                      (λ i r≢i → reflexive (1M-∁-diag r≢i) ⟨ ∗-cong ⟩ refl) ⟩
      Σ[ i ← 0 …< n ∥ ∁′ (≟ r) ] 0# ∗ A [ i , c ]
        ≈⟨ sym $ Σ.distrˡ _ 0# (0 …< n ∥ ∁′ (≟ r)) ⟩
      0# ∗ (Σ[ i ← 0 …< n ∥ (∁′ (≟ r)) ] A [ i , c ])
        ≈⟨ proj₁ zero _ ⟩
      0#
    ∎
```

The final lemma required to prove ⊗-identityˡ is filter, which states that the interval 0 …< n contains each number smaller than n exactly once.

```
  filter : ∀ r → 0 …< n ∥ ≟ r ≡ L.[ r ]
  filter r = ordinals-filter z≤n (bounded r)
```

```
⊗-identityˡ : ∀ A → 1M ⊗ A ≈ A
⊗-identityˡ A = λ r c →
  begin
    (1M ⊗ A) [ r , c ]
{- 4.27 -} ≡⟨ lookup∘tabulate r c ⟩
    Σ[ i ← 0 …< n ] 1M [ r , i ] * A [ i , c ]
{- 4.28 -} ≈⟨ Σ.split-P _ (0 …< n) (≟ r) ⟩
    Σ[ i ← 0 …< n ‖ ≟ r ]       1M [ r , i ] * A [ i , c ] +
    Σ[ i ← 0 …< n ‖ ∁′ (≟ r) ] 1M [ r , i ] * A [ i , c ]
{- 4.29 -} ≈⟨ ≡-step A r c ⟨ +-cong ⟩ ≢-step A r c ⟩
    Σ[ i ← 0 …< n ‖ ≟ r ] A [ i , c ] + 0#
{- 4.30 -} ≈⟨ proj₂ +-identity _ ⟩
    Σ[ i ← 0 …< n ‖ ≟ r ] A [ i , c ]
{- 4.31 -} ≡⟨ P.cong (Σ-syntax (λ i → A [ i , c ]))
                      (filter r) ⟩
    A [ r , c ] + 0#
{- 4.32 -} ≈⟨ proj₂ +-identity _ ⟩
    A [ r , c ]
  ∎
```

The proof of right-identity works in a similar way, and is omitted here. It is included in the Agda source of the project.

```
⊗-isMonoid : IsMonoid _≈_ _⊗_ 1M
⊗-isMonoid = record
  { isSemigroup = record
    { isEquivalence = ≈-isEquivalence
    ; assoc       = ⊗-assoc
    ; •-cong      = ⊗-cong
    }
  ; identity     = ⊗-identityˡ , ⊗-identityʳ
  }
```

4.5 Distributivity laws

Here we prove that _⊗_ right-distributes over _⊕_. The proof of left-distributivity works in a similar way, and is omitted in this report.

This proof shows that $A \otimes (B \oplus C) \approx (A \otimes B) \oplus (A \otimes C)$. Most of the proof is concerned with unfolding the definition of _⊕_ and _⊗_. The crucial step in the proof is the application of the left-distributivity law of the underlying semiring in inner followed by the use of Σ.merge in distr, splitting the sum into two.

$$
\begin{aligned}
(A \otimes (B \oplus C))_{r,c} &\approx \sum_{i \leftarrow 0 \ldots < n} A_{r,i}(B_{i,c} + C_{i,c}) && (4.33)\\
&\approx \sum_{i \leftarrow 0 \ldots < n} (A_{r,i}\, B_{i,c}) + (A_{r,i}\, C_{i,c}) && (4.34)\\
&\approx \left(\sum_{i \leftarrow 0 \ldots < n} A_{r,i}\, B_{i,c} \right) + \left(\sum_{i \leftarrow 0 \ldots < n} A_{r,i}\, C_{i,c} \right) && (4.35)\\
&\approx (A \otimes B)_{r,c} + (A \otimes C)_{r,c} && (4.36)\\
&\approx ((A \otimes B) \oplus (A \otimes C))_{r,c} && (4.37)
\end{aligned}
$$

inner proves that $\sum_{i \leftarrow 0 \ldots < n} A_{r,i}(B_{i,c} + C_{i,c}) \approx \sum_{i \leftarrow 0 \ldots < n} (A_{r,i}\, B_{i,c}) + (A_{r,i}\, C_{i,c})$ (4.34):

```
inner : ∀ {A B C : M} r c i →  A [ r , i ] ∗ (B ⊕ C) [ i , c ] ≈
                                A [ r , i ] ∗ B [ i , c ] + A [ r , i ] ∗ C [ i , c ]
inner {A} {B} {C} r c i = begin
  A [ r , i ] ∗ (B ⊕ C) [ i , c ]
    ≈⟨ refl ⟨ ∗−cong ⟩ reflexive (lookup∘tabulate i c) ⟩
  A [ r , i ] ∗ (B [ i , c ] + C [ i , c ])
    ≈⟨ proj₁ distrib _ _ _ ⟩
  A [ r , i ] ∗ B [ i , c ] + A [ r , i ] ∗ C [ i , c ] ∎
```

It is used in the distributivity proof as follows:

```
⊗−distrOverˡ−⊕ : ∀ A B C → A ⊗ (B ⊕ C) ≈ (A ⊗ B) ⊕ (A ⊗ C)
⊗−distrOverˡ−⊕ A B C = λ r c →
  begin
    (A ⊗ (B ⊕ C)) [ r , c ]
{- 4.33 -}  ≡⟨ lookup∘tabulate r c ⟩
    Σ[ i ← 0 ...< n ] A [ r , i ] ∗ (B ⊕ C) [ i , c ]
{- 4.34 -}  ≈⟨ Σ.cong (0 ...< n) P.refl (inner r c)⟩
    Σ[ i ← 0 ...< n ] (  A [ r , i ] ∗ B [ i , c ] +
      A [ r , i ] ∗ C [ i , c ])
{- 4.35 -}  ≈⟨ sym $ Σ.merge _ _ (0 ...< n) ⟩
    Σ[ i ← 0 ...< n ] A [ r , i ] ∗ B [ i , c ] +
    Σ[ i ← 0 ...< n ] A [ r , i ] ∗ C [ i , c ]
{- 4.36 -}  ≡⟨ P.sym $ lookup∘tabulate r c ⟨ P.cong₂ _+_ ⟩ lookup∘tabulate r c ⟩
    (A ⊗ B) [ r , c ] + (A ⊗ C) [ r , c ]
{- 4.37 -}  ≡⟨ P.sym $ lookup∘tabulate r c ⟩
    ((A ⊗ B) ⊕ (A ⊗ C)) [ r , c ]
  ∎
```

4.6 Summary

Taking all the lemmas in this Chapter together, we have shown that square matrices over a semiring again form a semiring:

```
M-isSemiring : IsSemiring _≈_ _⊕_ _⊗_ 0M 1M
M-isSemiring = record
  { isSemiringWithoutAnnihilatingZero = record
    { +-isCommutativeMonoid  = ⊕-isCommutativeMonoid
    ; *-isMonoid             = ⊗-isMonoid
    ; distrib                = ⊗-distrOverˡ-⊕ , ⊗-distrOverʳ-⊕
    }
  ; zero                     = M-zeroˡ , M-zeroʳ
  }

M-Semiring : Semiring c ℓ
M-Semiring = record
  { Carrier     = M
  ; _≈_         = _≈_
  ; _+_         = _⊕_
  ; _*_         = _⊗_
  ; 0#          = 0M
  ; 1#          = 1M
  ; isSemiring  = M-isSemiring
  }
```

This concludes our proof that the square matrices over a semiring form a semiring.

Chapter 5

Conclusions

We introduced the concept of a *big operator* and discussed its importance in many areas of mathematics in Chapter 1. We argued that Agda, the dependently typed programming language and proof assistant used in this project, emphasises readability of proofs. Using a small example, we demonstrated a proof style called *equational reasoning* which allows Agda users to write formal proofs that resemble pen-and-paper mathematics.

In Chapter 2 we provided a tutorial-style introduction to Agda and theorem proving using dependent types in general. Predicates and relations, provability, decidability, type inhabitation and setoids were introduced by example.

We discussed design decisions and described the implementation of our library in Chapter 3. We argued that the evaluation of a big operator is best modelled as a map from the index list into the carrier of a monoid, followed by a fold over the resulting list using the monoid's binary operator. In addition to big operators, our library implements notation and reasoning principles for intervals and filters.

A proof of the theorem "square matrices over a semiring form a semiring" was presented in Chapter 4. It includes a small matrix module and exercises all three components of our library (big operators, filters, intervals).

Success criterion

This project's success criterion was the construction of a set of reusable libraries that make it possible to write a readable proof of the theorem "square matrices over a semiring form a semiring" in Agda.[1] This criterion has been met:

- We have implemented a library for expressing and reasoning about big operators in an algebra-centric way.

- Additionally, modules for expressing and reasoning about matrices, intervals of natural numbers and filters have been implemented.

- The components of the library (big operators, intervals and filters) are independent but can easily be combined. All three are written in idiomatic Agda, and have been designed to interoperate with the Agda standard library.

[1] The wording in the proposal was "any square matrix over a semiring is a semiring", which is incorrect: one square matrix by itself does not constitute a semiring. The intended meaning was "square matrices over a semiring form a semiring".

- Propositions involving big operators can be written in a notation that resembles pen-and-paper mathematics using special syntax defined in each component.
- We have proved that square matrices over a semiring form a semiring (see Chapter 4). The proof demonstrates the use of all four components of our library.

In our project proposal we claimed that any implementation of big operators would depend on "a notion of enumerable finite sets". Experiments with prototype implementations using finite sets showed that this additional layer of abstraction only complicated the implementation with no tangible benefits. In Section 3.2.1 we argue that lists are the most idiomatic way to represent the collection of indices of a big operator.

5.1 Related work

In this section, we relate our project to previous implementations of big operators in the proof assistants Isabelle and Coq.

Isabelle

Isabelle's HOL library contains two definitions related to big operators.

- The module `Groups_Big` defines a Sigma-notation for big operators. Here, sets represent the collection of indices, and the underlying structure is assumed to be a commutative monoid (see Section 3.1 for how the representation of the index domain and the algebraic structure are related).
- The `List` theory defines a function `listsum`, which exactly corresponds to the function `crush` defined in Section 3.2.3: given a monoid, it performs a right-fold over the list, combining elements using the monoid's operator. If the list is empty, the identity element of the monoid is returned. No syntax definitions for `listsum` is provided.

There has been a large effort to formalise set theory in Isabelle, so building on top of sets as fundamental building block is a natural path to take. Since lists have more structure than sets, we argue that defining big operators over lists rather than sets is the more flexible, unifying approach (see Section 3.2.1).

Coq

The approach taken in Coq's `bigop` module,[2] which is part of the Mathematical Components library (and formerly of SSReflect), provided much inspiration for this project.[3] Internally, the library works in a very similar way to ours: in a monoid, `fold` (defined in Section 3.2.3) is equivalent to `reducebig`, the big operator evaluation function in `bigop`

[2]A clickable version of the source code for the `bigop` module can be accessed via `http://ssr.msr-inria.inria.fr/doc/mathcomp-1.5/MathComp.bigop.html`.

[3]The source code and documentation of both SSReflect and the Mathematical Components library for Coq can be found here: `http://ssr.msr-inria.inria.fr/`.

(see Appendix D.3 for a formal proof). One big difference is that there is no restriction on the underlying structure—`reducebig` does not require a monoid.

Lemmas, on the other hand, are structured in a similar way to our project: sections of the module assume certain properties of the underlying structure, and prove lemmas about big operators based on those assumptions.

The `bigop` module is more comprehensive than our library. It provides a lot of flexibility in terms of notation, and allows users to define big operators over sets, finite types and enumerable predicates (all of which are converted to lists internally).

Agda

Gustafsson and Pouillard (2014) formalises foldable containers in dependent type systems. It describes *exploration functions*, a class of folds of the same form as our big operator evaluation function `fold`. In contrast to our implementation, the type of exploration functions does not require their arguments to be monoids but "when proving properties about explorations, the monoid laws will have to be assumed as well" (Gustafsson and Pouillard 2014, p. 3). Various exploration functions are defined which extract information about the domain type. *Exploration transformers* like filters map exploration functions to exploration functions. An inductive reasoning principle for exploration functions is established. Finally the paper defines a way of exploring dependent Sigma- and Pi-types.

As the terminology suggests, the goal of this paper is to build a set of primitives for exploring types and their properties. It does not define a notation for big operators, nor does it provide reasoning principles like the ones discussed in Section 3.5.

5.2 Future work

We intend to make our library publicly available in the near future. We would like to extend it to include lemmas about more algebraic structures. As a simple example, in an *abelian group* (a commutative monoid with inverses for each carrier element) the following holds:

$$\left(\prod_i f(i)\right) \cdot \left(\prod_i \frac{1}{f(i)}\right) = 1$$

It would be interesting to see what other properties can be lifted from different underlying algebraic structures.

A different but related idea would be to attempt to write an automated solver for big operators using a metaprogramming technique called *proof by reflection* (Walt and Swierstra 2013). Agda's standard library already contains solvers for monoids and rings.[4] We expect finding a normal form for arbitrary big operator expressions to be the main challenge in such an effort to automate proofs involving big operators.

[4]Eric Mertens wrote a number of solvers for different algebraic structures in Agda, available here: `https://github.com/glguy/my-agda-lib/tree/master/Algebra`.

Bibliography

Bertot, Yves et al. (2008). “Canonical Big Operators”. In: *Theorem Proving in Higher Order Logics*. Ed. by Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar. Berlin; Heidelberg: Springer, pp. 86–101.

Coquand, Thierry (1992). “Pattern matching with dependent types”. In: *Proceedings of the 1992 Workshop on Types for Proofs and Programs*. Båstad.

Danielsson, Nils Anders and Ulf Norell (2011). “Parsing Mixfix Operators”. In: *Implementation and Application of Functional Languages*. Ed. by Sven-Bodo Scholz and Olaf Chitil. Lecture Notes in Computer Science 5836. Berlin; Heidelberg: Springer, pp. 80–99.

Gibbons, Jeremy and Bruno C. d. S. Oliveira (2009). “The essence of the Iterator pattern”. In: *Journal of Functional Programming* 19.3, p. 377.

Girard, Jean-Yves (1972). “Interprétation fonctionnelle et élimination des coupures de l’arithmétique d’ordre supérieur”. Doctoral thesis. Paris: Université Paris 7.

Gondran, Michel (2008). *Graphs, Dioids and Semirings: New Models and Algorithms*. In collab. with Michel Minoux. Operations Research/Computer Science Interfaces 41. New York: Springer. 383 pp.

Graham, Ronald L. (1994). *Concrete mathematics: a foundation for computer science*. In collab. with Donald Ervin Knuth and Oren Patashnik. 2nd ed. Reading, Mass: Addison-Wesley. 657 pp.

Gustafsson, Daniel and Nicolas Pouillard (2014). “Foldable containers and dependent types”. Submitted for publication in the 16th International Symposium on Principles and Practice of Declarative Programming.

Howard, William A. (1980). “The Formulæ-as-Types Notion of Construction”. In: *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Ed. by Jonathan P. Seldin and Roger Hindley. London; New York: Academic Press, pp. 479–490.

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

Marlow, Simon, ed. (2010). *Haskell 2010 Language Report*.

Meertens, Lambert (1996). “Calculate polytypically!” In: *Programming Languages: Implementations, Logics, and Programs*. Ed. by Herbert Kuchen and S. Doaitse Swierstra. Lecture Notes in Computer Science 1140. Springer Berlin Heidelberg, pp. 1–16.

Milner, Robin, ed. (1997). *The definition of Standard ML, revised edition.* Cambridge, Mass: MIT Press. 114 pp.

Norell, Ulf (2009). “Dependently Typed Programming in Agda”. In: *Advanced Functional Programming.* Ed. by Pieter Koopman, Rinus Plasmeijer, and Doaitse Swierstra. Lecture Notes in Computer Science 5832. Berlin; Heidelberg: Springer, pp. 230–266.

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

Sørensen, Morten H. and Pawel Urzyczyn (2006). *Lectures on the Curry-Howard Isomorphism.* First edition. Vol. 149. Studies in Logic and the Foundation of Mathematics. Amsterdam; Boston: Elsevier. 460 pp.

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. Lecture Notes in Computer Science 8241. Berlin; Heidelberg: Springer, pp. 157–173.

Appendix A

Predicate example

As an example of a more involved predicate that uses propositional equality in its definition, Collatz n provides evidence that if we keep iterating the function f (defined below), starting with f (suc n), eventually the result will be the number one.

$$f(n) = \begin{cases} n/2 & \text{if } n \equiv 0 \pmod 2 \\ 3n+1 & \text{if } n \equiv 1 \pmod 2 \end{cases}$$

We can provide evidence for this property by giving a natural number i together with a proof that iter f (suc n) i ≡ 1. Bundling a value and evidence for a property of that value together is the constructive version of the existential quantifier. The record type Σ defined in Section 2.1.4 is exactly what we require: Σ ℕ (λ i → iter (suc n) i ≡ 1). Since the type of the value (ℕ) is unambiguous from the context, we can define yet another shortcut for Σ where this type is inferred by Agda:

```
∃ : ∀ {a b} {A : Set a} → (A → Set b) → Set (a ⊔ b)
∃ = Σ _
```

This lets us define a type ∃ λ x → ..., which reads naturally as "there exists an x such that ...". We now have all the building blocks to write the predicate Collatz:

```
Collatz : ℕ → Set
Collatz n = ∃ λ i → iter f (suc n) i ≡ 1
  where
    f : ℕ → ℕ
    f n with n mod 2
    f n | zero       = n div 2
    f n | suc zero   = suc (3 * n)
    f n | suc (suc ())

    iter : ∀ {A} → (A → A) → A → ℕ → A
    iter f x zero     = x
    iter f x (suc i)  = iter f (f x) i
```

The function iter f x i simply applies f to x, i times:

$$\text{iter}\ f\ x\ i = \underbrace{(f \circ f \circ \cdots \circ f)}_{i\ \text{times}}\ x$$

In the definition of f, there is one piece of syntax that we have not come across so far: with lets us pattern match against the result of evaluating an arbitrary expression and updates the local context with any new type information gleaned from that pattern match. The return value of f depends on whether n is divisible by two. The expression n mod 2 gives the remainder of dividing n by two, the result of which is either zero or suc zero. Here the absurd pattern is required—if we leave it out, the totality checker complains. It indicates that there is no m such that n mod 2 equals suc (suc m).

Let's look at a few examples of Collatz n. With $n = 0$, we have to give an i such that the result of applying f to suc 0 evaluates to 1. But we already have suc 0 = 1, so we do not need to apply f at all and $i = 0$.

Note that in all these examples, the second element of the pair, which represents the proof iter (suc n) $i \equiv 1$, is just refl: given i, the type checker can evaluate the iteration and figure out that the equality holds.

```
Collatz‿0+1 : Collatz 0
Collatz‿0+1 = 0 , refl
```

With n = suc 1, the remainder after division by two is zero, so we do the division and get to 1 in one iteration: $i = 1$.

```
Collatz‿1+1 : Collatz 1
Collatz‿1+1 = 1 , refl
```

For n = suc 3, we divide by two twice to get to 1, giving $i = 2$.

```
Collatz‿3+1 : Collatz 3
Collatz‿3+1 = 2 , refl
```

As the following diagram shows, we need to apply f seven times to get from suc 2 to 1:

$$\begin{tikzcd}
\mathbf{3} \arrow[r, "\text{mod } 2{=}1", "\times 3{+}1"'] & 10 \arrow[r, "\text{mod } 2{=}0", "/2"'] & 5 \arrow[r, "\text{mod } 2{=}1", "\times 3{+}1"'] & 16 \arrow[r, "\text{mod } 2{=}0", "/2"'] & 8 \\
 & \cdots \arrow[r, "\text{mod } 2{=}0", "/2"'] & 4 \arrow[r, "\text{mod } 2{=}0", "/2"'] & 2 \arrow[r, "\text{mod } 2{=}0", "/2"'] & \mathbf{1}
\end{tikzcd}$$

```
Collatz‿2+1 : Collatz 2
Collatz‿2+1 = 7 , refl
```

Appendix B

Gauss formula

This Chapter presents a proof of the Gauss formula and a variation thereof for odd natural numbers. Both proofs use Σ-syntax and lemmas from the Bigop module. In the second proof, the predicate Odd and filters are used.

B.1 Gauss formula

In this Section, we show a pen-and-paper proof of the equation

$$2 \cdot \sum_{i=0}^{n} i = n \cdot (n + 1)$$

followed by a formal proof using definitions and lemmas from the Bigop module. The proof proceeds induction over n. The base case holds trivially as

$$2 \cdot \sum_{i=0}^{0} i = 0 = 0 \cdot (0 + 1)$$

The induction hypothesis is

$$2 \cdot \sum_{i \leftarrow 0 \ldots n} i = n \cdot (n + 1)$$

and the induction step works as follows:

$$\begin{aligned}
2 \cdot \sum_{i \leftarrow 0 \ldots 1+n} i &= 2 \cdot \left(\left(\sum_{i \leftarrow 0 \ldots n} i\right) + 1 + n\right) && \text{(B.1)}\\
&= \left(2 \cdot \sum_{i \leftarrow 0 \ldots n} i\right) + (2 \cdot (1 + n)) && \text{(B.2)}\\
&= n \cdot (1 + n) + 2 \cdot (1 + n) && \text{(B.3)}\\
&= 2 \cdot (1 + n) + n \cdot (1 + n) && \text{(B.4)}\\
&= (2 + n) \cdot (1 + n) && \text{(B.5)}\\
&= (1 + n) \cdot (2 + n) && \text{(B.6)}
\end{aligned}$$

In Agda, using Σ-syntax, the theorem

$$\forall n. \quad 2 \cdot \sum_{i=0}^{n} i = n \cdot (n + 1)$$

is expressed as

$$\forall\, n \to 2 * (\Sigma[\, i \leftarrow 0 \ldots n\,]\, i) \equiv n * (\mathsf{suc}\ n)$$

Proof by natural number induction over n translates to pattern matching on this argument in Agda. The base case is $n = \mathsf{zero}$; the induction step is given as the right-hand side of the pattern $\mathsf{suc}\ n$.

In the induction step, we use equational reasoning (see Section 2.4.3) to transform the equation step by step. Each step is annotated with the corresponding equation in the proof shown above. The lemmas used to justify the transformation are:

$\mathsf{proj}_1\ \mathsf{distrib}$	:	$\forall\, x\, y\, z \to x * (y + z) \equiv x * y + x * z$
$+\!-\!\mathsf{comm}$	:	$\forall\, x\, y \to x + y \equiv y + x$
$*\!-\!\mathsf{comm}$	:	$\forall\, x\, y \to x * y \equiv y * x$
lemma	:	$\Sigma[\, i \leftarrow 0 \ldots \mathsf{suc}\ n\,]\, i \equiv \Sigma[\, i \leftarrow 0 \ldots n\,]\, i + \mathsf{suc}\ n$

In step (C.3), the induction hypothesis proof n is used.

```
 proof : ∀ n → 2 * (Σ[ i ← 0 … n ] i) ≡ n * (suc n)
 proof zero = P.refl – trivial base case
 proof (suc n) =
   begin
     2 * (Σ[ i ← 0 … suc n ] i)
{- C.1 -} ≡⟨ P.cong (_*_ 2) lemma ⟩
     2 * (Σ[ i ← 0 … n ] i + suc n)
{- C.2 -} ≡⟨ proj₁ distrib 2 (Σ[ i ← 0 … n ] i) (suc n) ⟩
     2 * (Σ[ i ← 0 … n ] i) + 2 * suc n
{- C.3 -} ≡⟨ P.cong₂ _+_ (proof n) P.refl ⟩
     n * suc n + 2 * suc n
{- C.4 -} ≡⟨ +−comm (n * suc n) (2 * suc n) ⟩
     2 * suc n + n * suc n
{- C.5 -} ≡⟨ P.sym (proj₂ distrib (suc n) 2 n) ⟩
     (2 + n) * suc n
{- C.6 -} ≡⟨ *−comm (2 + n) (suc n) ⟩
     suc n * (suc (suc n))
   ∎
```

B.2 Odd Gauss formula

A variant of the Gauss formula is the equation

$$\forall n. \sum_{\substack{i \leftarrow 0 \ldots 2n \\ i \text{ odd}}} i = n^2$$

which using Σ-syntax and the filter function _∥_ can be written as follows in Agda:

$$\forall\, n \to \Sigma[\, i \leftarrow 0 \;...\!+\; \mathsf{suc}\ (n + n) \parallel \mathsf{odd}\ ]\ i \approx n * n$$

The lemma extract brings the list into a form more amenable to induction. It states that the list of odd numbers from zero up to but not including $2n + 3$ equals the list of odd numbers from zero up to but not including $2n + 1$ with $2n + 1$ appended to it.

In the first step (A) the auxiliary lemma 3suc is applied to rewrite suc (suc n + suc n) to suc (suc (suc ($n + n$))). Next (B) upFrom–last extracts the last element of the list 0 ...+ suc (suc ($n + n$)). Step (C) uses last–no and

$$\mathsf{even{\to}\neg odd}\ (\mathsf{ss{-}even}\ (\mathsf{2n{-}even}\ n))\ :\ \neg\, \mathsf{Odd}\ (\mathsf{suc}\ (\mathsf{suc}\ (n + n)))$$

to show that suc (suc ($n + n$)) is filtered out. In step (D) we again extract the last element of the list, suc ($n + n$). Lastly (E), we use last–yes and

$$\mathsf{even{+}1}\ (\mathsf{2n{-}even}\ n)\ :\ \mathsf{Odd}\ (\mathsf{suc}\ (n + n))$$

to show that suc ($n + n$ is odd, which allows us to take this element out of the filter.

```
  extract : ∀ n →  0 ... (suc n + suc n) ∥ odd ≡
                   0 ... (n + n) ∥ odd ∷ʳ suc (n + n)
  extract n =
     begin
        0 ... (suc n + suc n) ∥ odd
{- A -}  ≡⟨ P.cong (flip _∥_ odd ∘ _..._ 0) (+−suc (suc n) n) ⟩
        0 ... (suc (suc (n + n))) ∥ odd
{- B -}  ≡⟨ P.cong (flip _∥_ odd) (upFrom−last 0 (suc (suc (n + n)))) ⟩
        0 ... (suc (n + n)) ∷ʳ suc (suc (n + n)) ∥ odd
{- C -}  ≡⟨ last−no  (0 ... (suc (n + n))) (suc (suc (n + n))) odd
                     (even→¬odd (ss−even (2n−even n))) ⟩
        0 ... (suc (n + n)) ∥ odd
{- D -}  ≡⟨ P.cong (flip _∥_ odd) (upFrom−last 0 (suc (n + n))) ⟩
        0 ... (n + n) ∷ʳ suc (n + n) ∥ odd
{- E -}  ≡⟨ last−yes  (0 ... (n + n)) (suc (n + n)) odd
                      (even+1 (2n−even n)) ⟩
        0 ... (n + n) ∥ odd ∷ʳ suc (n + n)
     ∎
```

The proof of the odd Gauss equation again works by natural number induction on n, with a trivial base case for zero. In the induction step, we first rewrite the index list using extract (F). Then Σ.last is used to move the last element of the list out of the sum (G). In step (H) we use the induction hypothesis. The remainder of the proof just rewrites the algebraic expression into the required form.

```
  proof : ∀ n → Σ[ i ← 0 ... (n + n) ∥ odd ] i ≡ n ∗ n
  proof zero = P.refl
  proof (suc n) =
    begin
      Σ[ i ← 0 ... (suc n + suc n) ∥ odd ] i
{- F -}  ≡⟨ P.cong (fold id) (extract n)⟩
      Σ[ i ← 0 ... (n + n) ∥ odd ∷ʳ suc (n + n) ] i
{- G -}  ≡⟨ Σ.last id (suc (n + n)) (0 ... (n + n) ∥ odd) ⟩
      Σ[ i ← 0 ... (n + n) ∥ odd ] i + suc (n + n)
{- H -}  ≡⟨ +−cong (proof n) refl ⟩

      n ∗ n + suc (n + n)  ≡⟨ +−cong (refl {x = n ∗ n}) (sym (+−suc n n)) ⟩
      n ∗ n + (n + suc n)  ≡⟨ sym $ +−assoc (n ∗ n) n (suc n) ⟩
      (n ∗ n + n) + suc n  ≡⟨ +−cong (+−comm (n ∗ n) _) refl ⟩
      suc n ∗ n + suc n    ≡⟨ +−comm (suc n ∗ n) _ ⟩
      suc n + suc n ∗ n    ≡⟨ +−cong (refl {x = suc n}) (∗−comm (suc n) n) ⟩
      suc n ∗ suc n
    ∎
```

Appendix C

Binomial theorem

In this Chapter, we use the Bigop module to prove a special case of the binomial theorem (see, for example, equation 5.13 on page 163 in Graham (1994)):

$$\sum_{k \leftarrow 0 \ldots n-1} \binom{n}{k} \cdot x^k = (1 + x)^n$$

We present a pen-and-paper proof first and then translate it into Agda.

C.1 Pen-and-paper proof

The proof works by natural number induction on n. The base case with $n = 0$ is trivial as $\binom{n}{0} \cdot x^0 = 1 = (1 + x)^n$. The induction step proceeds as follows:

$$\begin{aligned}
\sum_{k \leftarrow 0 \ldots n+1} \binom{n+1}{k} \cdot x^k &= 1 + \sum_{k \leftarrow 1 \ldots n+1} \binom{n+1}{k} \cdot x^k && \text{(C.1)}\\
&= 1 + \left(\sum_{k \leftarrow 0 \ldots n} \binom{n}{k} \cdot x^{k+1} + \sum_{k \leftarrow 0 \ldots n} \binom{n}{k+1} \cdot x^{k+1} \right) && \text{(C.2)}\\
&= \sum_{k \leftarrow 0 \ldots n} \binom{n}{k} \cdot x^{k+1} + \left(1 + \sum_{k \leftarrow 0 \ldots n} \binom{n}{k+1} \cdot x^{k+1} \right) && \text{(C.3)}\\
&= x \cdot \sum_{k \leftarrow 0 \ldots n} \binom{n}{k} \cdot x^k + \sum_{k \leftarrow 0 \ldots n} \binom{n}{k} \cdot x^k && \text{(C.4)}\\
&= x \cdot \sum_{k \leftarrow 0 \ldots n} \binom{n}{k} \cdot x^k + 1 \cdot \sum_{k \leftarrow 0 \ldots n} \binom{n}{k} \cdot x^k && \text{(C.5)}\\
&= (x + 1) \cdot \sum_{k \leftarrow 0 \ldots n} \binom{n}{k} \cdot x^k && \text{(C.6)}\\
&= (1 + x)^{1+n} && \text{(C.7)}
\end{aligned}$$

Here the last step uses the induction hypothesis, $\sum_{k \leftarrow 0 \ldots n} \binom{n}{k} \cdot x^k = (1 + x)^n$.

C.2 Definitions

Since the Agda standard library does not currently define exponentials and binomials for natural numbers, we start by writing down their inductive definitions:

```
_^_ : ℕ → ℕ → ℕ
x ^ 0      = 1
x ^ suc n  = x * x ^ n

_choose_ : ℕ → ℕ → ℕ
_      choose  0      = 1
0      choose  suc k  = 0
suc n  choose  suc k  = n choose k + n choose (suc k)
```

Additionally we define a shorthand f for the general form of the function we will be manipulating within the sum:

```
f : ℕ → ℕ → ℕ → ℕ
f x n k = n choose k * x ^ k
```

C.3 Lemmas

In this Section we prove the lemmas used in the final proof of the binomial theorem. split justifies the step from (D.1) to (D.2)

$$\sum_{k \leftarrow 1 \dots n+1} \binom{n+1}{k} \cdot x^k = \left(\sum_{k \leftarrow 0 \dots n} \binom{n}{k} \cdot x^{k+1} \right) + \left(\sum_{k \leftarrow 0 \dots n} \binom{n}{k+1} \cdot x^{k+1} \right)$$

by shifting the values of its index list down by one and splitting the sum into two. In the actual proof, the addition with 1 is taken care of using reflexivity and congruence of addition.

```
split : ∀ x n →     Σ[ k ← 1 … suc n ] (suc n) choose k ∗ x ^ k ≡
                    Σ[ k ← 0 … n ] n choose k ∗ x ^ (suc k)
                    + Σ[ k ← 0 … n ] n choose (suc k) ∗ x ^ (suc k)
split x n =
  begin
    Σ[ k ← 1 … suc n ] (suc n) choose k ∗ x ^ k
      ≡⟨ sym $ P.cong (fold (f x (suc n))) (upFrom−suc 0 (suc n)) ⟩
    Σ[ k ← map suc (0 … n) ] (suc n) choose k ∗ x ^ k
      ≡⟨ sym $ Σ.map′ (f x (suc n)) suc (0 … n) (λ _ _ → refl) ⟩
    Σ[ k ← 0 … n ] (n choose k + n choose (suc k)) ∗ x ^ (suc k)
      ≡⟨ Σ.cong {f = λ k → (n choose k + n choose (suc k)) ∗ x ^ (suc k)}
                (0 … n) P.refl
                (λ k → distribʳ (x ^ (suc k)) (n choose k) _) ⟩
    Σ[ k ← 0 … n ] ( n choose k ∗ x ^ (suc k)
                   + n choose (suc k) ∗ x ^ (suc k))
      ≡⟨ sym $ Σ.merge (λ k → n choose k ∗ x ^ (suc k)) (λ k → f x n (suc k))
                       (0 … n) ⟩
    Σ[ k ← 0 … n ] n choose k ∗ x ^ (suc k)
      + Σ[ k ← 0 … n ] n choose (suc k) ∗ x ^ (suc k)
  ∎
```

+−reorder simply re-arranges three summands, as it is done in the pen-and-paper proof between (D.2) and (D.3):

$$1+\left(\sum_{k\leftarrow 0\ldots n}\binom{n}{k}\cdot x^{k+1}+\sum_{k\leftarrow 0\ldots n}\binom{n}{k+1}\cdot x^{k+1}\right)=\sum_{k\leftarrow 0\ldots n}\binom{n}{k}\cdot x^{k+1}+\left(1+\sum_{k\leftarrow 0\ldots n}\binom{n}{k+1}\cdot x^{k+1}\right)$$

```
+−reorder : ∀ x y z → x + (y + z) ≡ y + (x + z)
+−reorder x y z =
  begin
    x + (y + z)  ≡⟨ sym $ +−assoc x y z ⟩
    (x + y) + z  ≡⟨ +−cong (+−comm x y) refl ⟩
    (y + x) + z  ≡⟨ +−assoc y x z ⟩
    y + (x + z)
  ∎
```

We also prove ∗−reorder, which proves that the same transformation holds for multiplication. This auxiliary lemma is used in left−distr (below).

```
∗−reorder : ∀ x y z → x ∗ (y ∗ z) ≡ y ∗ (x ∗ z)
∗−reorder x y z =
  begin
    x ∗ (y ∗ z)  ≡⟨ sym $ ∗−assoc x y z ⟩
    (x ∗ y) ∗ z  ≡⟨ ∗−cong (∗−comm x y) refl ⟩
    (y ∗ x) ∗ z  ≡⟨ ∗−assoc y x z ⟩
    y ∗ (x ∗ z)
  ∎
```

The lemma left−distr uses ∗−reorder and the left-distributivity law for sums (Σ.distrˡ) to pull a factor x out of the exponential in the sum. It provides justification for going from the left-hand side of the outer addition in (D.3) to the left-hand side of the addition in (D.4):

$$\sum_{k \leftarrow 0 \dots n} \binom{n}{k} \cdot x^{k+1} = x \cdot \sum_{k \leftarrow 0 \dots n} \binom{n}{k} \cdot x^{k}$$

```
left−distr : ∀ x n →  Σ[ k ← 0 … n ] n choose k ∗ x ^ (suc k) ≡
                      x ∗ (Σ[ k ← 0 … n ] n choose k ∗ x ^ k)
left−distr x n =
  begin
    Σ[ k ← 0 … n ] n choose k ∗ x ^ (suc k)
      ≡⟨ Σ.cong (0 … n) P.refl (λ k → ∗−reorder (n choose k) x (x ^ k)) ⟩
    Σ[ k ← 0 … n ] x ∗ (n choose k ∗ x ^ k)
      ≡⟨ sym $ Σ.distrˡ (f x n) x (0 … n) ⟩
    x ∗ (Σ[ k ← 0 … n ] n choose k ∗ x ^ k)
  ∎
```

The auxiliary lemma choose−lt is equivalent to $p < q \rightarrow p$ choose $q \equiv 0$, but this is the form in which it is used in choose−suc. The keyword mutual allows choose−lt to be defined in terms of choose−lt′ and vice versa.

```
mutual
  choose−lt : ∀ m n → n choose (suc m + n) ≡ 0
  choose−lt m zero     = P.refl
  choose−lt m (suc n)  = choose−lt′ m n ⟨ +−cong ⟩ choose−lt′ (suc m) n

  choose−lt′ : ∀ m n → n choose (m + suc n) ≡ 0
  choose−lt′ m n = begin
    n choose (m + suc n)  ≡⟨ P.refl ⟨ P.cong₂ _choose_ ⟩ +−suc m n ⟩
    n choose suc (m + n)  ≡⟨ choose−lt m n ⟩
    0 ∎
```

choose−lt is required for choose−suc, which in turn is used in shift (below).

```
choose−suc : ∀ x n →  Σ[ k ← 0 … n ] n choose (suc k) ∗ x ^ (suc k) ≡
                      Σ[ k ← 1 … n ] n choose k ∗ x ^ k
choose−suc x n  =
  begin
    Σ[ k ← 0 … n ] n choose (suc k) ∗ x ^ (suc k)
      ≡⟨ Σ.map′ (f x n) suc (0 … n) (λ _ _ → refl) ⟩
    Σ[ k ← map suc (0 … n) ] n choose k ∗ x ^ k
      ≡⟨ P.cong (fold $ f x n) (upFrom−suc 0 (suc n)) ⟩
    Σ[ k ← 1 … suc n ] n choose k ∗ x ^ k
      ≡⟨ P.cong (fold $ f x n) (upFrom−last 1 n) ⟩
    Σ[ k ← (1 … n) ∷ʳ (suc n) ] n choose k ∗ x ^ k
      ≡⟨ Σ.last (f x n) (suc n) (1 … n) ⟩
    (Σ[ k ← 1 … n ] n choose k ∗ x ^ k) + n choose (suc n) ∗ x ^ (suc n)
      ≡⟨ +−cong  (refl {x = Σ[ k ← 1 … n ] f x n k})
                 (choose−lt 0 n ⟨ ∗−cong ⟩ refl ⟨ trans ⟩ zeroˡ n) ⟩
    (Σ[ k ← 1 … n ] n choose k ∗ x ^ k) + 0
      ≡⟨ proj₂ +−identity _ ⟩
    Σ[ k ← 1 … n ] n choose k ∗ x ^ k
  ∎
```

Our final lemma shift justifies the equality between the right-hand side of the outer addition in (D.3) and the right-hand side of the outer addition in (D.4):

$$\left(1 + \sum_{k\leftarrow 0\ldots n} \binom{n}{k+1} \cdot x^{k+1}\right) = \sum_{k\leftarrow 0\ldots n} \binom{n}{k} \cdot x^{k}$$

```
shift : ∀ x n →  1 + Σ[ k ← 0 … n ] n choose (suc k) ∗ x ^ (suc k)
                 ≡ Σ[ k ← 0 … n ] n choose k ∗ x ^ k
shift x n =
  begin
    1 + Σ[ k ← 0 … n ] n choose (suc k) ∗ x ^ (suc k)
      ≡⟨ (refl {x = 1}) ⟨ +−cong ⟩  (choose−suc x n) ⟩
    1 + Σ[ k ← 1 … n ] n choose k ∗ x ^ k
      ≡⟨ refl ⟩
    Σ[ k ← 0 ∷ (1 … n) ] n choose k ∗ x ^ k
      ≡⟨ P.cong (fold $ f x n) (upFrom−head 0 n) ⟩
    Σ[ k ← 0 … n ] n choose k ∗ x ^ k
  ∎
```

C.4 Proof

The following Agda proof is annotated by the corresponding steps in the pen-and-paper proof presented at the beginning of the chapter.

```
  proof : ∀ x n → Σ[ k ← 0 … n ] n choose k ∗ x ^ k ≡ (suc x) ^ n
  proof x zero  = refl
  proof x (suc n) =
    begin
      1 + Σ[ k ← 1 … suc n ] (suc n) choose k ∗ x ^ k
{- D.1 -}  ≡⟨ refl {x = 1} ⟨ +-cong ⟩ (split x n) ⟩
      1 + (  Σ[ k ← 0 … n ] n choose k ∗ x ^ (suc k)
           + Σ[ k ← 0 … n ] n choose (suc k) ∗ x ^ (suc k))
{- D.2 -}  ≡⟨ +-reorder 1 (Σ[ k ← 0 … n ] n choose k ∗ x ^ (suc k)) _ ⟩
      Σ[ k ← 0 … n ] n choose k ∗ x ^ (suc k)
      + (1 + Σ[ k ← 0 … n ] n choose (suc k) ∗ x ^ (suc k))
{- D.3 -}  ≡⟨ left-distr x n ⟨ +-cong ⟩ shift x n ⟩
      x ∗  (Σ[ k ← 0 … n ] n choose k ∗ x ^ k) +
           (Σ[ k ← 0 … n ] n choose k ∗ x ^ k)
{- D.4 -}  ≡⟨ refl {x = x ∗ _} ⟨ +-cong ⟩ sym (proj₁ ∗-identity _) ⟩
      x ∗  (Σ[ k ← 0 … n ] n choose k ∗ x ^ k) +
      1 ∗  (Σ[ k ← 0 … n ] n choose k ∗ x ^ k)
{- D.5 -}  ≡⟨ sym $ distribʳ _ x 1 ⟩
      (x + 1) ∗ (Σ[ k ← 0 … n ] n choose k ∗ x ^ k)
{- D.6 -}  ≡⟨ +-comm x 1 ⟨ ∗-cong ⟩ proof x n ⟩
      (suc x) ^ (suc n)
    ∎
```

This proves the Binomial theorem in Agda.

Appendix D

Additional proofs

D.1 Arrow-pair isomorphism

We prove that the types $A \times B$ and $A \to B$ are isomorphic by defining the functions curry : $(A \to B \to C) \to (A \times B \to C)$ and uncurry : $(A \times B \to C) \to (A \to B \to C)$ and proving that they are inverses of each other. curry and uncurry are easily defined:[1]

```
curry : ∀ {a b c} {A : Set a} {B : Set b} {C : Set c} → (A → B → C) → (A × B → C)
curry f (x , y) = f x y

uncurry : ∀ {a b c} {A : Set a} {B : Set b} {C : Set c} → (A × B → C) → (A → B → C)
uncurry f x y = f (x , y)
```

In order to show that curry and uncurry constitute an isomorphism, we prove that they are inverses of each other:

```
uncurry∘curry : ∀ {a b c} {A : Set a} {B : Set b} {C : Set c}
                (f : A → B → C) (x : A) (y : B) →
                f x y ≡ uncurry (curry f) x y
uncurry∘curry f x y = refl

curry∘uncurry : ∀ {a b c} {A : Set a} {B : Set b} {C : Set c}
                (f : A × B → C) (x : A) (y : B) →
                f (x , y) ≡ curry (uncurry f) (x , y)
curry∘uncurry f x y = refl
```

Thus A → B → Set and A × B → Set are isomorphic and we can use them interchangeably.

[1]It is also possible to write curry and uncurry for dependent pairs (Σ).

D.2 Equality of left and right fold

foldr≡foldl shows that for any list, the right- and left-fold over that list evaluate to the same value if the binary operator and identity of a monoid are supplied as the first and second argument in either case. The proof relies on two lemmas. foldl−cong shows that foldl is congruent in its second argument:

```
foldl−cong : ∀ x y xs → x ≈ y → foldl _•_ x xs ≈ foldl _•_ y xs
foldl−cong x y []        x≈y = x≈y
foldl−cong x y (z :: zs)  x≈y = foldl−cong (x • z) (y • z) zs (•−cong x≈y refl)
```

foldl−step shows that a pre-multiplication using the monoid's binary operator can be factored into a left fold:

```
foldl−step : ∀ x xs → x • foldl _•_ ε xs ≈ foldl _•_ (ε • x) xs
foldl−step x [] = begin
  x • ε  ≈⟨ proj₂ identity x ⟩
  x      ≈⟨ sym (proj₁ identity x) ⟩
  ε • x  ∎
foldl−step x (y :: ys) = begin
  x • foldl _•_ (ε • y) ys    ≈⟨ •−cong refl (sym (foldl−step y ys)) ⟩
  x • (y • foldl _•_ ε ys)    ≈⟨ sym (assoc x y _) ⟩
  (x • y) • foldl _•_ ε ys    ≈⟨ foldl−step (x • y) ys ⟩
  foldl _•_ (ε • (x • y)) ys ≈⟨ foldl−cong  (ε • (x • y)) (ε • x • y) ys
                                             (sym (assoc ε x y)) ⟩
  foldl _•_ (ε • x • y) ys    ∎
```

Finally the proof the left- and right-folds over a monoid are equivalent proceeds by list induction:

```
foldr≡foldl : (xs : List R) → foldr _•_ ε xs ≈ foldl _•_ ε xs
foldr≡foldl []          = refl
foldr≡foldl (x :: xs)   = begin
  x • foldr _•_ ε xs    ≈⟨ •−cong refl (foldr≡foldl xs) ⟩
  x • foldl _•_ ε xs    ≈⟨ foldl−step x xs ⟩
  foldl _•_ (ε • x) xs  ∎
```

D.3 Extensional equality with reducebig

The `bigop` module for Coq defines an evaluation function for big operators called `reducebig`.[2] Here we translate it into an Agda function reducebig and show that its result is equal to evaluating our library's big operator function, fold.

```
reducebig : ∀ {i p} {I : Set i} {P : Pred I p} →
            (I → R) → Decidable P → List I → R
reducebig f p = foldr (λ i acc → if ⌊ p i ⌋ then f i • acc else acc) ε

equivalent : ∀ {i p} {I : Set i} {P : Pred I p} →
             (f : I → R) (p : Decidable P) (is : List I) →
             reducebig f p is ≡ fold f (is ∥ p)
equivalent f p []        = P.refl
equivalent f p (i ∷ is)  with p i
... | yes pi     = P.cong (_•_ (f i)) (equivalent f p is)
... | no ¬pi     = equivalent f p is
```

[2]A syntax-highlighted and clickable version of this module's source code is available here: `http://ssr.msr-inria.inria.fr/doc/mathcomp-1.5/MathComp.bigop.html`.