Formalizing the translation method in Agda

Bjarki Ágúst Guðmundsson

Thesis of 60 ECTS credits
Master of Science (M.Sc.) in Computer Science

June 2017

Formalizing the translation method in Agda

by

Bjarki Ágúst Guðmundsson

Thesis of 60 ECTS credits submitted to the School of Computer Science
at Reykjavík University in partial fulfillment
of the requirements for the degree of
Master of Science (M.Sc.) in Computer Science

June 2017

Supervisors:

Anders Claesson, Supervisor
Professor, University of Iceland, Iceland

Henning Ulfarsson, Supervisor
Assistant Professor, Reykjavík University, Iceland

Examiners:

Antonis Achilleos, Examiner
Postdoctoral Researcher, Reykjavík University, Iceland

Brent Yorgey, Examiner
Assistant Professor, Hendrix College, USA

Copyright
Bjarki Ágúst Guðmundsson
June 2017

Formalizing the translation method in Agda

Bjarki Ágúst Guðmundsson

June 2017

Abstract

If P and Q are sets of combinatorial objects, the translation method, introduced by Wood and Zeilberger (2009), allows one to turn an algebraic proof of the identity $|P| = |Q|$ into a bijection between P and Q. We give a formalized implementation of the translation method in the programming language Agda. In contrast to the implementation previously given by Wood and Zeilberger, the bijections produced by our implementation are formally verified, making our implementation more robust. We also take advantage of the fact that Agda is a proof assistant, allowing users of our implementation to use the existing facilities provided by Agda for developing proofs. In particular, converting an existing algebraic proof for use in our implementation is often straightforward.

We prove that the expressions used in our implementation form a commutative semiring. Based on this we implement a semiring solver, allowing one to apply the translation method automatically to certain identities. We also show how cancellation procedures, introduced by Feldman and Propp (1995), can be used to give meaning to subtraction, division and k-th roots in the translation method, and we implement the cancellation procedure that represents subtraction. Finally we give a philosophical discussion about the inner workings of the translation method, and use that to count the number of "natural" bijections for an identity considered by Wood and Zeilberger.

Að formgera þýðingaraðferðina í Agda

Bjarki Ágúst Guðmundsson

júní 2017

Útdráttur

Ef P og Q eru mengi af fléttufræðilegum hlutum, þá gefur þýðingaraðferðin, sem sett var fram af Wood og Zeilberger (2009), umbreytingu á algebrulegri sönnun á jöfnunni $|P| = |Q|$ í gagntæka vörpun á milli P og Q. Við gefum formlega útfærslu á þýðingaraðferðinni í forritunarmálinu Agda. Í samanburði við útfærsluna sem var áður gefin af Wood og Zeilberger, þá eru gagntæku varpanirnar sem eru framleiddar af okkar útfærslu formlega staðfestar, sem gerir okkar útfærslu stöðugri. Við nýtum okkur einnig að Agda getur veitt aðstoð við að útbúa sannanir, og gerum við notendum útfærslu okkar kleift að nota þá aðstöðu sem Agda veitir til þess. Sér í lagi er oft einfalt að undirbúa gamlar algebrulegar sannanir til notkunnar í útfærslu okkar.

Við sönnum að segðirnar sem eru notaðar í útfærslunni okkar mynda víxlinn hálfbaug. Með þetta í huga útfærum við svokallaðan leysara fyrir hálfbauga, sem gerir notendum kleift að framkvæma þýðingaraðferðina sjálfvirkt á ákveðnar jöfnur. Við sýnum líka hvernig svokölluð afturköllunarferli, sem sett voru fram af Feldman og Propp (1995), geta verið nýtt til að gefa frádrætti, deilingu, og k-tu rótum þýðingu í þýðingaraðferðinni, og útfærum við afturköllunarferlið sem samsvarar frádrætti. Að lokum gefum við heimspekilega umfjöllun um hvernig þýðingaraðferðin virkar, og notum það til að telja fjölda „náttúrulegra“ gagntækra varpana fyrir jöfnu sem var skoðuð af Wood og Zeilberger.

To my family.

Acknowledgements

First and foremost I want to thank Anders Claesson for suggesting an interesting topic to work on, agreeing—with a very short notice—to become my thesis advisor, as well as finding time to meet me on a weekly basis.

I also want to thank Henning Ulfarsson for suggesting problems to work on, as well as collaborating with me on a few of them, even though I ended up doing something else for my thesis.

I want to thank the awesome people at the School of Computer Science, in particular Hallgrímur Arnalds, Magnús Már Halldórsson, Sigrún María Ammendrup, Sigurbjörg Ásta Hreinsdóttir and Yngvi Björnsson, for help and support with many different things.

Many thanks to Bernhard Linn Hilmarsson for keeping me busy; it would have been so much easier to finish this thesis without you. Also thanks to Unnar Freyr Erlendsson, Arnar Bjarni Arnarson, Garðar Andri Sigurðsson and Hannes Kristján Hannesson for helping me keep him at bay.

And finally, I'm very grateful to Sara Dögg Sigurðardóttir for helping me prepare for my thesis defence.

This research was partially supported by grant 141761-051 from the Icelandic Research Fund.

Contents

Acknowledgements **ix**

Contents **xi**

List of Figures **xiii**

List of Tables **xv**

1 Introduction **1**
- 1.1 Definitions . . . 2
- 1.2 The translation method . . . 3
- 1.3 Agda and propositions as types . . . 17
  - 1.3.1 Algebraic data types . . . 17
  - 1.3.2 Functions . . . 19
  - 1.3.3 Dependent types . . . 20
  - 1.3.4 Standard library . . . 30
  - 1.3.5 Syntax . . . 31

2 The translate module **33**
- 2.1 Expressions . . . 34
- 2.2 Equivalence of expressions . . . 38
- 2.3 Tools . . . 44
- 2.4 Using the translate module . . . 48
- 2.5 A semiring solver . . . 51
  - 2.5.1 Heuristics for the solver . . . 64
- 2.6 On the number of natural bijections . . . 67
- 2.7 Cancellation . . . 70
  - 2.7.1 Cancellation of addition . . . 71
  - 2.7.2 Other cancellation procedures . . . 76
- 2.8 Further examples . . . 77

3 Conclusions and future work **87**

Bibliography **91**

A Setting up the translate module **93**

List of Figures

2.1 A depiction of the semiring solver. Given expressions f and g, they are normalized, and then the user is expected to provide a proof that the normalized versions are equivalent. 54
2.2 An example of the cancellation procedure for addition. On the left we have the bijection $f : A \sqcup B \to C \sqcup D$, denoted with solid line segments, and the bijection $g : B \to D$, denoted with dashed line segments. On the right is a trace of the procedure starting at x_1, which ends at y_3. The resulting bijection $A \cong C$ maps $x_1 \mapsto y_3$, $x_2 \mapsto y_2$ and $x_3 \mapsto y_1$. 72

List of Tables

1.1 Standard interpretations of common expressions. Here a and b can be lifted to A and B, respectively. 4
1.2 Common identities and their lifted versions. Here a, b and c can be lifted to A, B and C, respectively. 7
1.3 Pairs of binary strings mapped to quaternary strings using the bijection Φ_n. 11
1.4 The bijections T_4 and T_5. 14

2.1 A part of the bijection produced by automatically proving the identity $(a+b)^2 = a^2 + 2ab + b^2$, with `nothing` denoted as `zero` and `just nothing` denoted as `one` for brevity. 65
2.2 The bijections `bijection (thrice' {4})` and `bijection (thrice' {5})`. . 68
2.3 A few samples from `bijection (part n k)`, using the textual representation given by `show`. 82

Chapter 1

Introduction

Combinatorics is the study of all kinds of discrete mathematical objects. One of the classical questions in Combinatorics is: given two finite sets of discrete objects, do they have the same cardinality? When that is the case, or sometimes even towards proving that, combinatorialists seek a bijection between the two sets. That is, a way to pair up the elements of the two sets, so that each element is paired with a distinct element of the other set. This is clearly possible if and only if the two sets have the same cardinality.

When such bijections are defined in a natural manner—for some definition of "natural"—they will often shed a light on the underlying structure of the sets, giving intuition as to *why* they have the same cardinality. This intuition will then serve as a stepping stone towards further study of these, as well as related sets of discrete objects.

Many combinatorialists base their whole careers on finding "elegant" bijections, or bijections that have nice properties. Doing so is often considered an art form, and, in the past, has been something that required both creativity and ingenuity. It should therefore come as no surprise that combinatorialists have sought ways to make it easier to come up with bijections. While this is still an unexplored field, there have been at least three types of approaches so far:

Bijection machines: Parameterized bijections that work for a general class of problems, with Remmel's bijection machine being one example [27]. It works for a large class of identities involving integer partitions, and in many cases the bijections produced coincide with the bijections that had previously been discovered by combinatorialists.

Semi-automatic methods: Methods that help derive a bijection for a given problem, but require some assistance from the combinatorialist. An example of this is Wood and Zeilberger's translation method [37]. It takes an algebraic proof of an identity, and turns it into a bijection.

Fully automatic methods: Some have even envisioned completely automatic methods for finding bijections, e.g. Chow [7]. As far as we know, these ideas have yet to be implemented.

In the context of semi-automatic methods, the translation method has proved to be a very helpful tool for coming up with bijections, and has had many successful applications [14, 28–31]. Unfortunately the method is somewhat loosely defined, and its

current implementation, in Maple, has some caveats, as we will later see. This makes it both cumbersome and clumsy to apply the method in its current form.

To remedy this, we propose a formalization of the translation method in the programming language Agda, along with some extensions that make it easier to apply. Our hope, then, is that this enables more combinatorialists to apply the translation method to their own problems.

The thesis is organized as follows. This chapter will give an introduction to both the translation method and the programming language Agda. Readers already familiar with Agda may skip most of that section, although they may want to take note of our notation, as well as our definition of bijections. Our main body of work is then presented in Chapter 2. In Sections 2.1–2.4 we present the translate module—our implementation of the translation method in Agda. We also describe some simple helper tools for the module, as well as give a few examples. In Section 2.5 we describe a solver that can prove certain identities automatically, simplifying the use of our translate module in those cases. In Section 2.6 we give a philosophical discussion of the translation method—looking at its inner workings—and use that to count natural bijections. In Section 2.7 we describe procedures that give meaning to subtraction, as well as a few other operations, in the translation method. In Section 2.8 we then give a few larger examples. Finally, in Chapter 3 we give some closing remarks, and suggest future work. But first, let's present a few definitions.

1.1 Definitions

If A and B are sets, $A \times B$ is their Cartesian product. If A and B are disjoint, $A \sqcup B$ is their disjoint union. We will later give meaning to the disjoint union for sets that are not disjoint. We denote the empty set with $\emptyset$.

Let $[n] = \{0, 1, \ldots, n-1\}$, i.e. the set of natural numbers below n. Let $\mathrm{id} : A \to A$ denote the identity function, $\mathrm{id}(x) = x$. We denote with $f : A \cong B$ the fact that f is a bijection between A and B. Clearly $\mathrm{id} : A \cong A$.

The Fibonacci numbers are given by the recurrence relation $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$, as well as the base cases $F_0 = 1$, $F_1 = 1$. The binomial coefficients, $\binom{n}{k}$, count the number of k-element subsets of $[n]$. The Bell numbers, B_n, count the number of set partitions of $[n]$, and the Stirling numbers of the second kind, $S(n, k)$, count the number of such partitions with exactly k parts.

Let $\mathbf{2}^n$ denote the set of binary strings of length n, and $\mathbf{4}^n$ denote the set of quaternary strings of length n. Let $\mathcal{F}_n$ denote the set of integer compositions of weight n that use only 1's and 2's as summands. We will represent such integer compositions as strings of 1's and 2's. Let $\binom{[n]}{k}$ denote the set of binary strings of length n with exactly k 1's. Let Par_n denote the set of set partitions of $[n]$, and Par_n^k denote the set of such set partitions with exactly k parts.

A commutative semiring is a set equipped with addition and multiplication that satisfies the following properties:

- Addition is commutative and associative, and has identity element 0.
- Multiplication is commutative and associative, and has identity element 1.

- Multiplication distributes over addition.
- Multiplication by 0 yields 0.

In other words, a commutative semiring is a commutative ring that does not necessarily have additive inverses.

As it plays a central role in our work, let us now present Wood and Zeilberger's translation method [37], albeit in a manner more relevant for our purposes.

1.2 The translation method

Say we have two sets of combinatorial objects, P and Q, and the goal is to find a bijection between them. Furthermore, and this will be crucial, we have an algebraic proof that $|P| = |Q|$. Clearly there exists at least one bijection between the two sets, as they have equally many elements, but we're not just looking for any bijection. As we mentioned above, we want a bijection that is "natural", in the sense that it preserves the underlying structure of the elements of P and Q, giving the combinatorialist more insight into *why* the two sets have equally many elements. Following Wood and Zeilberger, we will informally define such a bijection as a *natural bijection*.

To find this bijection, the translation method makes use of the algebraic proof that $|P| = |Q|$, which we assume the combinatorialist already has, "lifting" the proof to a bijection between P and Q. The hope is then that, if the algebraic proof was in some sense "natural", then a natural bijection will be produced. None of this is very formal, and we don't have the tools to make it so, but we hope that the examples throughout the rest of our discussion will make it clear that the translation method can, often with a pretty mechanical procedure, produce useful bijections that would otherwise have required some ingenuity to come up with.

Before showing how the translation method works, let us first define what we mean by "lifting".

Definition 1 (Lifting an expression)**.** Whenever n is an integer expression, and N is a set of combinatorial objects with $|N| = n$, we say that n can be *lifted* to N.

For example, the integer 2^k can be lifted to the set of binary strings of length k. Similarly, the integer F_k, the k-th Fibonacci number, can be lifted to the set of integer compositions of k that use only 1's and 2's as summands (and indeed, it can be proved that there are F_k of those). In this way lifting will be used to give a combinatorial interpretation of an expression.

One point that should be clarified about lifting is that it is inherently ambiguous, as a given expression can have multiple combinatorial interpretations. For example, 2^k can also be lifted to the set of natural numbers less than 2^k. This, however, turns out to be a pretty useless interpretation, as it doesn't give us much combinatorial structure to work with. But when such ambiguity arises, it should be clear from context which interpretation we want.

Table 1.1 shows standard interpretations of common expressions that we may encounter. Here are a couple of things that require further elaboration. First, when lifting $a + b$, one might wonder what should happen if A and B are not disjoint. To

Expression E	Lifted E
c	$[c]$
$a+b$	$A \sqcup B$
$a \cdot b$	$A \times B$
2^n	binary strings of length n: $\mathbf{2}^n$
F_n	integer compositions of n using only 1's and 2's: $\mathcal{F}_n$
$\binom{n}{k}$	binary strings of length n with exactly k 1's: $\binom{[n]}{k}$
B_n	set partitions of $[n]$: Par_n
$S(n,k)$	set partitions of $[n]$ containing exactly k parts: Par_n^k

Table 1.1: Standard interpretations of common expressions. Here a and b can be lifted to A and B, respectively.

circumvent this issue, we will label all the elements in A with L (for left) and all the elements in B with R (for right). That way, if A and B have elements in common, these elements will be distinct in $A \sqcup B$. Second, note that we do not show how to lift expressions such as $a - b$. This is because it is generally not straightforward to give a combinatorial interpretation of such an expression, in particular when $B \not\subseteq A$. The same holds for expressions such as a/b. It is nevertheless possible to deal with these expressions under special circumstances, and we will show how in Section 2.7.

With these interpretations we can now easily lift an expression such as $2 \cdot 2^{k-1}$ — it's just the set $\{0,1\} \times \mathbf{2}^{k-1}$, where $\mathbf{2}^n$ is the set of binary strings of length n. Now that we know how to lift expressions, we can extend the concept of lifting to identities.

Definition 2 (Lifting an identity)**.** If n and m are integers which can be lifted to N and M, respectively, and $f : N \to M$ is a bijection between N and M, then we say that the identity $n = m$ can be lifted to f.

Consider the identity $2^k = 2 \cdot 2^{k-1}$, when $k \geq 1$, which follows from the definition of exponentiation. As we have seen, 2^k can be lifted to $\mathbf{2}^k$, and $2 \cdot 2^{k-1}$ can be lifted to $\{0,1\} \times \mathbf{2}^{k-1}$. Now let $\mathrm{bin}_k : \mathbf{2}^k \to \{0,1\} \times \mathbf{2}^{k-1}$ be a function defined as follows, for all $k \geq 1$:

$$\mathrm{bin}_k(s) = \begin{cases} (0,t) & \text{if } s = 0t \\ (1,t) & \text{if } s = 1t \end{cases}$$

It is clear that bin_k forms a bijection between $\mathbf{2}^k$ and $\{0,1\} \times \mathbf{2}^{k-1}$, which are the lifted versions of 2^k and $2 \cdot 2^{k-1}$, respectively, so we can say that the identity $2^k = 2 \cdot 2^{k-1}$ can be lifted to the bijection bin_k. Thus, similar to what lifting did for expressions, lifting gives us a combinatorial interpretation of identities, but now in terms of bijections.

Lifting, both of expressions and identities, is what makes the translation method possible. Indeed, going back to the high level introduction of the translation method, where we sought a bijection between two sets of combinatorial objects, P and Q, assuming we have an algebraic proof that $|P| = |Q|$, we can now see that the translation method is just lifting the identity $|P| = |Q|$ to a bijection.

The last thing we need before being able to define the translation method are some building blocks—some commonly used lifted identities. If we consider the algebraic proof that $|P| = |Q|$, it may rely on facts such as

- the commutativity of addition, associativity of multiplication, or distributivity of multiplication over addition, or more generally the fact that the natural numbers, coupled with addition and multiplication, form a commutative semiring,
- definitions, such as $F_n = F_{n-1} + F_{n-2}$,
- congruence of addition and multiplication,
- that $=$ forms an equivalence relation, or even
- "smaller" versions of the identity being proved, by induction.

Each of these facts, interpreted as identities, can be lifted to a corresponding bijection. In particular, the fact that $=$ forms an equivalence relation is lifted to the fact that bijections form an equivalence relation, and the congruence of addition and multiplication are lifted to the congruence of disjoint union and Cartesian product on sets, respectively. Regarding lifting of semiring properties, here follow a few examples.

Example 3. Commutativity of addition can be stated as the identity $a+b = b+a$. If the expressions a and b can be lifted to the sets A and B, respectively, then the lifted identity we're looking for is a bijection $f : A \sqcup B \to B \sqcup A$. Luckily it is straightforward to make such a bijection (and this might even be the only valid definition):

$$f(x) = \begin{cases} y_{\mathsf{R}} & \text{if } x = y_{\mathsf{L}} \\ y_{\mathsf{L}} & \text{if } x = y_{\mathsf{R}} \end{cases}$$

Note here that we use L and R to mark which elements come from A and B, respectively, as discussed above.

Example 4. Associativity of multiplication can be stated as the identity $(a \cdot b) \cdot c = a \cdot (b \cdot c)$. If the expressions a, b and c can be lifted to the sets A, B and C, respectively, then the lifted identity we're looking for is a bijection $f : (A \times B) \times C \to A \times (B \times C)$. Again this is a straightforward exercise, and the following bijection works:

$$f((x, y), z) = (x, (y, z))$$

Example 5. Distributivity of multiplication over addition (from the left) can be stated as the identity $a \cdot (b+c) = a \cdot b + a \cdot c$. If the expressions a, b and c can be lifted to the sets A, B and C, respectively, then the lifted identity we're looking for is a bijection $f : A \times (B \sqcup C) \to (A \times B) \sqcup (A \times C)$. There's a little bit more going on here, but still easy to come up with the following bijection:

$$f(x, y) = \begin{cases} (x, z)_{\mathsf{L}} & \text{if } y = z_{\mathsf{L}} \\ (x, z)_{\mathsf{R}} & \text{if } y = z_{\mathsf{R}} \end{cases}$$

We have also seen some examples of how definitions, such as $2^k = 2 \cdot 2^{k-1}$, can be lifted to bijections. A similar, but slightly more elaborate example is that of the Fibonacci identity:

Example 6. The inductive definition of Fibonacci numbers is given by the identity $F_n = F_{n-1} + F_{n-2}$, where $n \geq 2$. The left and right sides of this equation can be lifted to $\mathcal{F}_n$ and $\mathcal{F}_{n-1} \sqcup \mathcal{F}_{n-2}$, respectively. Thus, we're looking for a bijection $f : \mathcal{F}_n \to \mathcal{F}_{n-1} \sqcup \mathcal{F}_{n-2}$. Noting that an integer composition, with only 1's and 2's as summands, must either end with a 1, following such a composition of weight $n-1$, or with a 2, following such a composition of weight $n-2$, gives us the following bijection:

$$f(s) = \begin{cases} t_{\mathsf{L}} & \text{if } s = t1 \\ t_{\mathsf{R}} & \text{if } s = t2 \end{cases}$$

So, as we have seen, lifting provides a very general framework of assigning a combinatorial interpretation to identities. The only thing that can be considered a special case, but only barely so, is the concept of induction, and we will see how the translation method deals with that later.

Table 1.2 shows these, as well as other common identities, and their lifted versions. We leave it as an exercise for the reader to find the precise bijections of the corresponding lifted type, but as with the above examples, they should be straightforward to find.

This finally brings us to the translation method.

Definition 7 (The translation method)**.** Given two sets of combinatorial objects, P and Q, as well as an algebraic proof that $|P| = |Q|$, the translation method allows one to lift the proof to a bijection between P and Q. The method proceeds as follows:

1. Decompose the algebraic proof into atomic proof steps. Most of the atomic proof steps will rely on one or more of the identities presented in Table 1.2.
2. Lift each atomic proof step to a bijection. Again, this will usually be a simple combination of one or more of the lifted identities, and their associated bijections, from Table 1.2. If the proof step is using a "smaller" identity of the form $|P| = |Q|$ by induction, then that identity can be lifted recursively using the translation method. If neither Table 1.2 nor recursion can be used, the proof step has to be lifted manually.
3. Compose the lifted bijections to obtain a bijection between P and Q.

To gain a better understanding of the translation method, let's dive straight into a couple of examples, starting with a relatively simple one.

Example 8. Let's consider the equality $2^n \cdot 2^n = 4^n$. We can prove this equality by induction as follows. When $n = 0$, we have $2^0 \cdot 2^0 = 4^0 \iff 1 \cdot 1 = 1$, which is true. When $n > 0$, we have

$$\begin{aligned}
2^n \cdot 2^n &= (2 \cdot 2^{n-1}) \cdot (2 \cdot 2^{n-1}) && \text{def. exponentiation} && (1)\\
&= ((2 \cdot 2^{n-1}) \cdot 2) \cdot 2^{n-1} && \text{associativity} && (2)\\
&= (2 \cdot (2 \cdot 2^{n-1})) \cdot 2^{n-1} && \text{commutativity} && (3)\\
&= ((2 \cdot 2) \cdot 2^{n-1}) \cdot 2^{n-1} && \text{associativity} && (4)\\
&= (2 \cdot 2) \cdot (2^{n-1} \cdot 2^{n-1}) && \text{associativity} && (5)\\
&= 4 \cdot (2^{n-1} \cdot 2^{n-1}) && 2 \cdot 2 = 4 && (6)\\
&= 4 \cdot 4^{n-1} && \text{induction} && (7)\\
&= 4^n && \text{def. exponentiation} && (8)
\end{aligned}$$

	Identity	Lifted identity
= reflexive	$a = a$	$A \cong A$
= symmetric	$\dfrac{a = b}{b = a}$	$\dfrac{A \cong B}{B \cong A}$
= transitive	$\dfrac{a = b \qquad b = c}{a = c}$	$\dfrac{A \cong B \qquad B \cong C}{A \cong C}$
+ commutative	$a + b = b + a$	$A \sqcup B \cong B \sqcup A$
$\cdot$ commutative	$a \cdot b = b \cdot a$	$A \times B \cong B \times A$
+ associative	$(a + b) + c = a + (b + c)$	$(A \sqcup B) \sqcup C \cong A \sqcup (B \sqcup C)$
$\cdot$ associative	$(a \cdot b) \cdot c = a \cdot (b \cdot c)$	$(A \times B) \times C \cong A \times (B \times C)$
$\cdot$ distributes over +	$a \cdot (b + c) = a \cdot b + a \cdot c$	$A \times (B \sqcup C) \cong (A \times B) \sqcup (A \times C)$
+ congruence	$\dfrac{a = c \qquad b = d}{a + b = c + d}$	$\dfrac{A \cong C \qquad B \cong D}{A \sqcup B \cong C \sqcup D}$
$\cdot$ congruence	$\dfrac{a = c \qquad b = d}{a \cdot b = c \cdot d}$	$\dfrac{A \cong C \qquad B \cong D}{A \times B \cong C \times D}$
+ identity	$a + 0 = a$	$A \sqcup \emptyset \cong A$
$\cdot$ identity	$a \cdot 1 = a$	$A \times [1] \cong A$
$\cdot$ zero	$a \cdot 0 = 0$	$A \times \emptyset \cong \emptyset$
Exponentiation	$a^n = a \cdot a^{n-1}$	$A^n \cong A \times A^{n-1}$
Fibonacci numbers	$F_n = F_{n-1} + F_{n-2}$	$\mathcal{F}_n \cong \mathcal{F}_{n-1} \sqcup \mathcal{F}_{n-2}$
Binomial coefficients	$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$	$\binom{[n]}{k} \cong \binom{[n-1]}{k-1} \sqcup \binom{[n-1]}{k}$
Stirling numbers	$S(n,k) = k \cdot S(n-1,k) + S(n-1,k-1)$	$\mathrm{Par}_n^k \cong ([k] \times \mathrm{Par}_{n-1}^k) \sqcup \mathrm{Par}_{n-1}^{k-1}$

Table 1.2: Common identities and their lifted versions. Here a, b and c can be lifted to A, B and C, respectively.

Hence, by induction, our equality holds.

Now notice that $2^n \cdot 2^n$ can be lifted to $\mathbf{2}^n \times \mathbf{2}^n$, i.e. the set of pairs of binary strings of length n, and 4^n can be lifted to $\mathbf{4}^n$, i.e. the set of quaternary strings of length n. Using the translation method we might be able to lift our equality to a bijection $\mathbf{2}^n \times \mathbf{2}^n \cong \mathbf{4}^n$ based on our above inductive proof.

Following the above definition of the translation method, the first step is to decompose our proof into atomic steps. In our case, we already have the atomic steps, and they are numbered 1–8 in the proof above. The next step is then to lift each of the atomic steps. Let's do that, in detail, for each of the eight steps:

1. Here we want to lift the identity $2^n \cdot 2^n = (2 \cdot 2^{n-1}) \cdot (2 \cdot 2^{n-1})$. This identity follows from applying the identity $2^n = 2 \cdot 2^{n-1}$ on each side of the multiplication sign, which is allowed because multiplication forms a congruence. Since the identity $2^n = 2 \cdot 2^{n-1}$ can be lifted to the bijection bin_n, and congruence of multiplication can be lifted to the (parameterized) bijection $\text{mult-cong}_{f,g}$, this step can be lifted to the bijection

$$\varphi_{n,1} = \text{mult-cong}_{\text{bin}_n,\text{bin}_n}$$

2. Here we want to lift the identity $(2 \cdot 2^{n-1}) \cdot (2 \cdot 2^{n-1}) = ((2 \cdot 2^{n-1}) \cdot 2) \cdot 2^{n-1}$. This is associativity of multiplication, which we know can be lifted to the bijection mult-assoc. But one has to be careful here, as the identity we have here is of the form $a \cdot (b \cdot c) = (a \cdot b) \cdot c$, but mult-assoc was defined in terms of the identity $(a \cdot b) \cdot c = a \cdot (b \cdot c)$. So the bijection we're looking for is the symmetry of mult-assoc, and thus this step can be lifted to

$$\varphi_{n,2} = \text{mult-assoc}^{-1}$$

3. Here we want to lift the identity $((2 \cdot 2^{n-1}) \cdot 2) \cdot 2^{n-1} = (2 \cdot (2 \cdot 2^{n-1})) \cdot 2^{n-1}$. This is commutativity of multiplication, but nested inside the topmost multiplication on the left side. Thus we can lift this step to $\text{mult-cong}_{f,g}$, with mult-comm on the left side (and id on the right side to leave it untouched):

$$\varphi_{n,3} = \text{mult-cong}_{\text{mult-comm},\text{id}}$$

4. Here we want to lift the identity $(2 \cdot (2 \cdot 2^{n-1})) \cdot 2^{n-1} = ((2 \cdot 2) \cdot 2^{n-1}) \cdot 2^{n-1}$. This is a nested application of associativity of multiplication, so similar to the last two steps, this step can be lifted to

$$\varphi_{n,4} = \text{mult-cong}_{\text{mult-assoc}^{-1},\text{id}}$$

5. Here we want to lift the identity $((2 \cdot 2) \cdot 2^{n-1}) \cdot 2^{n-1} = (2 \cdot 2) \cdot (2^{n-1} \cdot 2^{n-1})$. This is associativity of multiplication, and this time it is of the same form as mult-assoc. Hence this step can be lifted to

$$\varphi_{n,5} = \text{mult-assoc}$$

6. Here we want to lift the identity $(2 \cdot 2) \cdot (2^{n-1} \cdot 2^{n-1}) = 4 \cdot (2^{n-1} \cdot 2^{n-1})$. Here we're making use of the fact that $2 \cdot 2 = 4$, on the left side of the multiplication. We have not seen the identity $2 \cdot 2 = 4$ before, and it is not in Table 1.2, so we will have to lift it manually. Lifting the expressions on each side of the identity gives us the sets $[2] \times [2]$ and $[4]$, respectively, so we're looking for a bijection $f : [2] \times [2] \to [4]$. One can verify that the following is a valid bijection:

$$f(x,y) = \begin{cases} 0 & \text{if } x = 0 \text{ and } y = 0 \\ 1 & \text{if } x = 0 \text{ and } y = 1 \\ 2 & \text{if } x = 1 \text{ and } y = 0 \\ 3 & \text{if } x = 1 \text{ and } y = 1 \end{cases}$$

However, we could have easily permuted the four values in any way, for a total of 4! possible valid bijections.

When it is the case that there are multiple valid bijections, we usually try to pick the bijection that, in some sense, is the most natural. As this choice will have impact on which bijection the translation method produces, it is the hope that picking natural bijections in the intermediate steps will result in a natural bijection being produced.

Unfortunately there is no clear choice here that is more natural than the other. But in this case it turns out, as we will see, that the choice does not matter too much, so this bijection will do fine.

With this bijection, we can now lift this step to

$$\varphi_{n,6} = \text{mult-cong}_{f,\text{id}}$$

7. Here we want to lift the identity $4 \cdot (2^{n-1} \cdot 2^{n-1}) = 4 \cdot 4^{n-1}$. On the right side of the multiplication we are using the identity $2^{n-1} \cdot 2^{n-1} = 4^{n-1}$. Notice that this is precisely the identity we are using the translation method on, but with a smaller value of n. So, as suggested in the definition of the translation method, we will apply the translation method recursively on this identity to get a lifted bijection, which we'll call Φ_{n-1}. This step can then be lifted to

$$\varphi_{n,7} = \text{mult-cong}_{\text{id},\Phi_{n-1}}$$

8. Here we want to lift the identity $4 \cdot 4^{n-1} = 4^n$. The symmetry of this identity is $4^n = 4 \cdot 4^{n-1}$, which follows from the definition of exponentiation in Table 1.2. For completeness, the lifted bijection, $\text{quad}_k : \mathbf{4}^k \to [4] \times \mathbf{4}^{k-1}$, is as follows:

$$\text{quad}_k(s) = \begin{cases} (0,t) & \text{if } s = 0t \\ (1,t) & \text{if } s = 1t \\ (2,t) & \text{if } s = 2t \\ (3,t) & \text{if } s = 3t \end{cases}$$

We can lift this step to

$$\varphi_{n,8} = \text{quad}_n^{-1}$$

Now that we have lifted each step to a bijection, the next step is to compose these bijections to get the required bijection $\mathbf{2}^n \times \mathbf{2}^n \cong \mathbf{4}^n$:

$$\Phi_n = \varphi_{n,8} \circ \varphi_{n,7} \circ \varphi_{n,6} \circ \varphi_{n,5} \circ \varphi_{n,4} \circ \varphi_{n,3} \circ \varphi_{n,2} \circ \varphi_{n,1}$$

There are two issues with this construction, though. First, some of the identities used in the intermediate steps require that $n \geq 1$. Second, we are recursively using the translation method for one of the steps, but currently we have no base case. Both of these issues can be fixed by treating $n = 0$ as a base case. As there is a unique pair of binary strings of length 0 (and a unique quaternary string of length 0), this is trivial:

$$\Phi_0(\varepsilon, \varepsilon) = \varepsilon,$$

where ε represents the empty string.

Now we finally have the required bijection between pairs of binary strings of length n and quaternary strings of length n, namely Φ_n. Table 1.3 shows how the bijection maps some pairs of small binary strings, as well as a few random pairs of larger binary strings, to quaternary strings.

This relatively simple example illustrates a couple of points about the translation method. First, the translation method is very tedious—and outright impractical—to do by hand. Luckily it turns out that the tedious parts are very mechanical, and can thus be easily carried out by a computer. And second, the bijection produced by the translation method is not direct, but rather a composition of many simpler bijections. Thus the resulting bijection is not immediately useful, but is rather intended to be experimentally used by the combinatorialist to discover a direct bijection[1].

Let us now give one more example, although in less detail than the first one.

Example 9. Consider the equality $3 \cdot F_n = F_{n+2} + F_{n-2}$, when $n \geq 2$. We can prove this equality by induction as follows. When $n = 2$, we have $3F_2 = F_4 + F_0 \iff 3 \cdot 3 = 8 + 1$, which is true, and when $n = 3$, we have $3F_3 = F_5 + F_1 \iff 3 \cdot 5 = 13 + 2$, which is also true. When $n \geq 4$, we have

$$\begin{aligned}
3F_n &= 3(F_{n-1} + F_{n-2}) && \text{Fibonacci def.} && (1)\\
&= 3F_{n-1} + 3F_{n-2} && \text{distributivity} && (2)\\
&= (F_{n+1} + F_{n-3}) + (F_n + F_{n-4}) && \text{induction} && (3)\\
&= F_{n+1} + (F_{n-3} + (F_n + F_{n-4})) && \text{associativity} && (4)\\
&= F_{n+1} + ((F_n + F_{n-4}) + F_{n-3}) && \text{commutativity} && (5)\\
&= F_{n+1} + (F_n + (F_{n-4} + F_{n-3})) && \text{associativity} && (6)\\
&= F_{n+1} + (F_n + (F_{n-3} + F_{n-4})) && \text{commutativity} && (7)\\
&= (F_{n+1} + F_n) + (F_{n-3} + F_{n-4}) && \text{associativity} && (8)\\
&= F_{n+2} + F_{n-2} && \text{Fibonacci def.} && (9)
\end{aligned}$$

Hence, by induction, our equality holds.

Now notice that $3 \cdot F_n$ can be lifted to $[3] \times \mathcal{F}_n$, i.e. the set of pairs of integers in the set $\{0, 1, 2\}$ and integer compositions of n using only 1's and 2's, and $F_{n+2} + F_{n-2}$

[1] Indeed, if one stares at Table 1.3 for long enough, one should be able to conjecture, and then go on to prove, a direct bijection.

s	$\Phi(s)$
$(\varepsilon, \varepsilon)$	ε
$(0, 0)$	0
$(0, 1)$	2
$(1, 0)$	1
$(1, 1)$	3
$(00, 00)$	00
$(00, 01)$	02
$(00, 10)$	20
$(00, 11)$	22
$(01, 00)$	01
$(01, 01)$	03
$(01, 10)$	21
$(01, 11)$	23
$(10, 00)$	10
$(10, 01)$	12
$(10, 10)$	30
$(10, 11)$	32
$(11, 00)$	11
$(11, 01)$	13
$(11, 10)$	31
$(11, 11)$	33
$(0000101011, 0010111111)$	0020323233
$(0000101100, 1101010100)$	2202121300
$(0001110010, 0000000010)$	0001110030
$(0010000110, 1101110101)$	2212220312
$(0011011101, 0001010101)$	0013031303
$(0100000000, 0010011000)$	0120022000
$(0110000101, 0101001010)$	0312002121
$(0110100010, 1110000010)$	2330100030
$(1000011101, 1001101101)$	3002213303
$(1000101111, 0001010011)$	1002121133
$(1000110011, 0011010001)$	1022130013
$(1001000101, 0100011001)$	1201022103
$(1001001011, 1001111011)$	3003223033
$(1011011110, 0010101100)$	1031213310
$(1011100101, 0000000110)$	1011100321
$(1011111111, 1000101110)$	3011313331
$(1101100011, 0010111110)$	1121322231
$(1101100111, 0111000001)$	1323100113
$(1110000110, 0011100101)$	1132200312
$(1110110110, 0110111111)$	1330332332
$(1110111100, 0100010000)$	1310131100
$(1111010111, 0000001010)$	1111012131
$(1111101011, 1110100010)$	3331301031

Table 1.3: Pairs of binary strings mapped to quaternary strings using the bijection Φ_n.

can be lifted to $\mathcal{F}_{n+2} \sqcup \mathcal{F}_{n-2}$, i.e. the union of the set of such integer compositions of weights $n+2$ and $n-2$. Using the translation method we might be able to lift our equality to a bijection $[3] \times \mathcal{F}_n \cong \mathcal{F}_{n+2} \sqcup \mathcal{F}_{n-2}$ based on the inductive proof.

Again we will follow the definition of the translation method given above. With the atomic steps presented in the algebraic proof above, we will now lift each of them in the same manner as in the previous example:

$$\begin{aligned}
t_{n,1} &= \text{mult-cong}_{\text{id},\text{fib}_n} \\
t_{n,2} &= \text{distrib-left} \\
t_{n,3} &= \text{plus-cong}_{T_{n-1},T_{n-2}} \\
t_{n,4} &= \text{plus-assoc} \\
t_{n,5} &= \text{plus-cong}_{\text{id},\text{plus-comm}} \\
t_{n,6} &= \text{plus-cong}_{\text{id},\text{plus-assoc}} \\
t_{n,7} &= \text{plus-cong}_{\text{id},\text{plus-cong}_{\text{id},\text{plus-comm}}} \\
t_{n,8} &= \text{plus-assoc}^{-1} \\
t_{n,9} &= \text{plus-cong}_{\text{fib}_{n+2}^{-1},\text{fib}_{n-2}^{-1}}
\end{aligned}$$

Then, for $n \geq 4$, our bijection is the composition of these lifted bijections:

$$T_n = t_{n,9} \circ t_{n,8} \circ t_{n,7} \circ t_{n,6} \circ t_{n,5} \circ t_{n,4} \circ t_{n,3} \circ t_{n,2} \circ t_{n,1}$$

Next we need our base cases, namely T_2 and T_3. In the previous example it was trivial to give a base case, as there was only one possible bijection. However, here we have 3! and 6! possible bijections for T_2 and T_3, respectively, and it is not clear which one is the correct one. Moreover, the choice of base cases may have a significant effect on the bijection produced, so picking arbitrary base cases may not be good enough. Here we should recall our heuristic that, in order for the resulting bijection to be natural, the individual steps, and the base cases in particular, should be natural as well. Alas, this does not seem to help much, as it is not clear what is natural on such a small scale, where there is so little structure to work with.

Lucky for us, this exact problem was already discussed by Wood and Zeilberger in [37], where they also present the relevant base cases. We will use these base cases for now, and then in Section 2.6 we will discuss how they were chosen:

$$T_2(x,s) = \begin{cases} 1111_{\mathsf{L}} & \text{if } x = 0 \text{ and } s = 11 \\ 112_{\mathsf{L}} & \text{if } x = 0 \text{ and } s = 2 \\ 211_{\mathsf{L}} & \text{if } x = 1 \text{ and } s = 11 \\ 22_{\mathsf{L}} & \text{if } x = 1 \text{ and } s = 2 \\ 121_{\mathsf{L}} & \text{if } x = 2 \text{ and } s = 11 \\ \varepsilon_{\mathsf{R}} & \text{if } x = 2 \text{ and } s = 2 \end{cases}$$

$$T_3(x,s) = \begin{cases} 11111_{\mathsf{L}} & \text{if } x = 0 \text{ and } s = 111 \\ 1112_{\mathsf{L}} & \text{if } x = 0 \text{ and } s = 12 \\ 1121_{\mathsf{L}} & \text{if } x = 0 \text{ and } s = 21 \\ 2111_{\mathsf{L}} & \text{if } x = 1 \text{ and } s = 111 \\ 212_{\mathsf{L}} & \text{if } x = 1 \text{ and } s = 12 \\ 221_{\mathsf{L}} & \text{if } x = 1 \text{ and } s = 21 \\ 1211_{\mathsf{L}} & \text{if } x = 2 \text{ and } s = 111 \\ 122_{\mathsf{L}} & \text{if } x = 2 \text{ and } s = 12 \\ 1_{\mathsf{R}} & \text{if } x = 2 \text{ and } s = 21 \end{cases}$$

With these base cases we finally have a bijection $T_n : [3] \times \mathcal{F}_n \to \mathcal{F}_{n+2} \sqcup \mathcal{F}_{n-2}$, for $n \geq 2$. Table 1.4 shows both T_4 and T_5. Again, if one stares at the table for long enough, one should be able to come up with a conjecture for the direct bijection, namely that

$$T(x,s) = \begin{cases} 11s & \text{if } x = 0 \\ 2s & \text{if } x = 1 \\ 12t & \text{if } x = 2 \text{ and } s = 1t \\ t & \text{if } x = 2 \text{ and } s = 2t \end{cases}$$

As previously mentioned the translation method is best carried out by a computer rather than by hand. Wood and Zeilberger also noticed this, and proceeded to give an implementation for some of the more mechanical steps in Maple[2] [37]. To get a flavour of their approach we'll reproduce their proof of the identity from Example 9 as well as their Maple rendering of it. Their proof is like the one in Example 9 but it leaves out several steps:

$$\begin{aligned} 3F_n &= 3F_{n-1} + 3F_{n-2} && \text{Fibonacci def. and distributivity} \\ &= F_{n+1} + F_{n-3} + F_n + F_{n-4} && \text{induction (twice), etc.} \\ &= F_{n-2} + F_{n+2} && \text{Fibonacci def., commutativity, etc.} \end{aligned}$$

Their base cases coincide with ours, as mentioned above. To turn this inductive proof into a bijection they use the translation method, similar to what we do in Example 9. Each step in their translation, however, covers a larger part of the derivation. To turn each of these steps into bijections, in Maple code, they use the following helper functions:

`IdBij(S)`	The identity bijection on the set S.
`FibDef(n)`	The bijection from Example 6.
`AddBij(f,g)`	The bijection $A \sqcup C \to B \sqcup D$, where $f : A \to B$, $g : C \to D$, $A \cap C = \emptyset$ and $B \cap D = \emptyset$.
`MultBij(f,g)`	The bijection $A \times C \to B \times D$, where $f : A \to B$ and $g : C \to D$.
`InvBij(f)`	The inverse bijection f^{-1}.
`ComposBij(f,g)`	The bijection $f \circ g$.

As can be seen from the base cases in the code listing below, Wood and Zeilberger use lookup tables as a data type to represent bijections.

[2]Maple is a computer algebra system developed by Maplesoft.

(x,s)	$T(x,s)$
$(0,1111)$	111111_{L}
$(0,211)$	11211_{L}
$(0,121)$	11121_{L}
$(0,112)$	11112_{L}
$(0,22)$	1122_{L}
$(1,1111)$	21111_{L}
$(1,211)$	2211_{L}
$(1,121)$	2121_{L}
$(1,112)$	2112_{L}
$(1,22)$	222_{L}
$(2,1111)$	12111_{L}
$(2,211)$	11_{R}
$(2,121)$	1221_{L}
$(2,112)$	1212_{L}
$(2,22)$	2_{R}
$(0,11111)$	1111111_{L}
$(0,2111)$	112111_{L}
$(0,1211)$	111211_{L}
$(0,1121)$	111121_{L}
$(0,221)$	11221_{L}
$(0,1112)$	111112_{L}
$(0,212)$	11212_{L}
$(0,122)$	11122_{L}
$(1,11111)$	211111_{L}
$(1,2111)$	22111_{L}
$(1,1211)$	21211_{L}
$(1,1121)$	21121_{L}
$(1,221)$	2221_{L}
$(1,1112)$	21112_{L}
$(1,212)$	2212_{L}
$(1,122)$	2122_{L}
$(2,11111)$	121111_{L}
$(2,2111)$	111_{R}
$(2,1211)$	12211_{L}
$(2,1121)$	12121_{L}
$(2,221)$	21_{R}
$(2,1112)$	12112_{L}
$(2,212)$	12_{R}
$(2,122)$	1222_{L}

Table 1.4: The bijections T_4 and T_5.

```
Id7egBase1:=proc() local B:
B:=table([
        [1,[1]] = [1,1,1],
        [2,[1]] = [2,1],
        [3,[1]] = [1,2]
]);
B:
end:

Id7egBase2:=proc() local B:
B:= table([ [1,[1,1]] = [1,1,1,1],
            [1,[2]]   = [1,1,2],
            [2,[1,1]] = [2,1,1],
            [2,[2]]   = [2,2],
            [3,[1,1]] = [1,2,1],
            [3,[2]]   = []
]);
B:
end:

Id7eg:=proc(n) local B;
if n=1 then
        return Id7egBase1():
elif n=2 then
        return Id7egBase2():
elif n>2 then
        ## Induction Step ##
        # 3 f_n = 3 f_{n-1} + 3 f_{n-2}
        B[1]:=MultBij(IdBij({1,2,3}), FibDef(n));

        # 3 f_{n-1} + 3 f_{n-2} = f_{n+1} + f_{n-3} + f_n + f_{n-4}
        B[3]:=AddBij(Id7eg(n-1), Id7eg(n-2)):

        # f_{n-3} + f_{n-4} + f_n + f_{n+1} = f_{n-2} + f_{n+2}
        B[4]:=InvBij(AddBij(FibDef(n-2),FibDef(n+2))):

        #       3 f_n = 3 f_{n-1} + 3 f_{n-2}
        #             = f_{n+1} + f_{n-3} + f_n + f_{n-4}
        #             = f_{n-2} + f_{n+2}
        B[5]:=ComposeBij(B[4],ComposeBij(B[3],B[1])):
        return B[5]:
fi:
# function should never reach this point.
return FAIL:
end:
```

This code will produce the same bijection as we got in Example 9. There are, however, a few things to note about the implementation. First, the conversion from the algebraic proof to code is unintuitive. That may of course be inherent to the translation method,

but not necessarily. And second, as Maple is a dynamically-typed language, it is easy to make small mistakes without noticing it, possibly leading the combinatorialist on a wild goose chase after something that turns out is not even a true bijection.

These concerns, along with the fact that Maple is proprietary software[3], prompted us to make an alternative implementation of the translation method in the statically-typed programming language Haskell.

However, as we were making our own implementation, we discovered some further issues with their implementation:

- They implement their bijections as lookup tables, and do all their operations directly on them. This does simplify some parts of the translation, as properties like commutativity and associativity of addition become irrelevant, and can be skipped as we saw in their proof. This, unfortunately, breaks down in other places. In particular, they don't provide a way to change associativity of multiplication. In fact, they specifically ask users of their implementation to avoid multiplying three or more sets, as well as multiplying by zero, as it may "screw things up".

- When performing addition, their implementation fails with an error in the case that the sets are not disjoint. This means that one cannot use their implementation to translate identities as simple as $2 \cdot a = a + a$. This would be better handled by doing as we suggest above, and assign an `L` or `R` label to each element, depending on which set an element originates from.

- When defining bijections explicitly, as is the case when one defines base cases, there is no check in place that makes sure that this is indeed a bijection, or that it is a bijection between the correct sets. Taking this to the extreme, we don't even know if their implementations of `AddBij`, `MultBij`, etc. behave as advertised. This, again, could be disastrous to a combinatorialist, who might be hunting down something that is not a bijection due to a bug.

None of these issues are related to the fact that Maple is a dynamically-typed language, and thus would not be fixed by merely changing to a statically-typed language. In particular, even if a statically-typed language can differentiate between, say, binary strings and integer compositions, it will not be able to differentiate between, say, binary strings of length 5 and binary strings of length 10.

This last issue suggests using a programming language that supports *dependent types*: types that can depend on values. In that case, binary strings of length 5 could be a distinct type from binary strings of length 10. Using dependent types also opens up doors to some very interesting possibilities, such as formally proving that what we have is indeed a bijection!

So this is what we set out to do. We investigated two general-purpose programming languages that supported dependent types: Agda [21, 23] and Idris [5, 6]. We found Agda to be more mature, and thus had a better experience with it while implementing more complex features. And thus, we decided to write our own implementation of the translation method in Agda.

[3] And as far as we could see, Maplesoft does not provide a free version for students or academics.

Before going on to describe our implementation, let us first give a quick introduction to Agda, its syntax, concepts, and features.

1.3 Agda and propositions as types

Agda is a pure functional programming language, similar in many respects to Haskell, that supports dependent types. We will come back to what dependent types are in just a bit, as they play a major role in our use of Agda, but as a start we will note that they are an additional feature of Agda's type system. If we ignore that feature, what remains is a type system very similar to what is found in most strongly-typed functional programming languages, Haskell in particular.

We will now give an informal introduction to Agda, specifically to the parts that are most relevant to us, by giving a few examples. To make the introduction more compact, we will assume some familiarity with the programming language Haskell. Let's start with one of the most fundamental objects in Agda, which are the *algebraic data types*.

1.3.1 Algebraic data types

In Agda, and especially in our work, it is helpful to think of algebraic data types as sets (in the mathematical sense). Consider the set of booleans, which contains only the values `true` and `false`. This set can be defined as follows in Agda:

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

This can be read as follows: `Bool` is a set, `true` is an element (or an inhabitant) of the `Bool` set, and `false` is also an element of the `Bool` set.

In many other functional programming languages, the term `Set` is replaced with `Type`. In that case, this would be read as: `Bool` is a type with two constructors; `true` and `false`. In fact, the above mathematical perspective and this perspective, perhaps more from the computer science spectrum, are used interchangeably in Agda.

It is also possible to make inductive definitions of sets. As an example, consider the natural numbers. There are many ways to represent natural numbers in computers, with representations in base 2 (binary) being the most common. In Agda, it turns out that base 1 (unary, or Peano arithmetic) is a representation that is perhaps the most commonly useful. This representation can be defined inductively: `zero` is a natural number, and if `n` is a natural number, then `suc n` is also a natural number, specifically the successor of `n`. In Agda, this is defined as follows[4]:

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

Again, this can be read as follows: `ℕ` is a set with `zero` as an element, and if `n` is an element of `ℕ`, then `suc n` is also an element of `ℕ`. Alternatively we can think of `zero`

[4] Unicode is used extensively in Agda.

and `suc` as constructors: `zero` is a constructor that takes no arguments and gives us something of type ℕ, while `suc` is a constructor that takes one argument of type ℕ and gives us something of type ℕ.

With this data type, we can now easily—but perhaps a bit verbosely—represent natural numbers: 0 is `zero`, 1 is `suc zero`, 2 is `suc (suc zero)`, and so on.

Now, since it is possible to view algebraic data types as sets, one may wonder if it is possible to apply any of the usual set-theoretic operations, such as union, the Cartesian product, intersection or subtraction, to these sets. In Agda it turns out that the latter two operations are not meaningful a priori, as elements of different sets are inherently distinct. It is, however, possible to apply both the union and the Cartesian product operations. The results of applying these operations are, of course, sets, and so it makes sense to also represent these operations as algebraic data types.

In the Cartesian product of two sets `A` and `B`, each element is a pair `(a, b)`, with `a` and `b` elements of `A` and `B`, respectively. This set can be defined in Agda as follows:

```
data _×_ (A : Set) (B : Set) : Set where
  _,_ : A → B → A × B
```

Here `_×_` is the name of our Cartesian product set. Note that the set is parameterized by two other sets, `A` and `B`. These will be the two sets the Cartesian product is applied to. Thus `_×_ A B` denotes the set that is the Cartesian product of `A` and `B`.

Notice the peculiar usage of underscores in the definition. Agda supports something called mixfix notation. In mixfix notation, underscores in identifiers can be substituted with arguments that would otherwise be specified after the identifier in the more common prefix notation. Therefore, in this case, `_×_ A B` and `A × B` are interchangeable. One restriction, however, is that the different parts of the identifier must be surrounded by one or more spaces, e.g. `A×B` will actually be a distinct identifier.

Our Cartesian product only has one constructor, `_,_`, which takes as arguments one element from `A` and one element from `B`, in that order, and gives us something in their Cartesian product. This will often be specified as `(a , b)`, where `a` and `b` are elements of `A` and `B`, respectively. Again, note the spaces around the comma.

For the union operation, each element will either be from set `A` or from set `B`. This can be defined as follows in Agda:

```
data _⊔_ (A : Set) (B : Set) : Set where
  inj₁ : A → A ⊔ B
  inj₂ : B → A ⊔ B
```

Here we have two constructors; `inj₁` for elements from `A`, and `inj₂` for elements from `B`.

Now we have given a few examples of data types in Agda. One may wonder why we mentioned natural numbers, but not signed integers. In our work we will be making extensive use of natural numbers, avoiding signed integers for the most part. There are some obvious caveats related to that decision, such as the fact that we can't easily represent negative numbers, and that subtraction becomes more cumbersome. However, the reason we decided to do this, and indeed the reason natural numbers are

so commonly used in Agda, is that they are easily pattern matched, and are ideal for use in recursion/induction.

This brings us to another fundamental object in Agda: functions.

1.3.2 Functions

Functions in Agda, at their simplest, are very similar to functions in Haskell. Let's give a few examples. Say we want to make a function that takes in a natural number n, and returns the natural number $n + 3$. This would be implemented as follows:

```
incr-3 : ℕ → ℕ
incr-3 n = suc (suc (suc n))
```

More generally we can implement addition for natural numbers: given natural numbers a and b, return the natural number $a + b$. In order to do that we will need to employ *pattern matching*. Pattern matching is simply a case analysis on one of the arguments. For example, since a is an element of the set ℕ, we know that it is either `zero` or `suc a'`, for some `a'` which is also an element of ℕ. Addition is then implemented as follows:

```
_+_ : ℕ → ℕ → ℕ
_+_ zero b = b
_+_ (suc a') b = suc (a' + b)
```

That is, if a is 0, then $a + b$ is simply b. On the other hand, if a is the successor of a', then $a + b$ is the successor of $a' + b$, which can then be computed recursively.

It is also possible to perform pattern matching on multiple arguments at the same time. Consider the following function which checks if two natural numbers are equal:

```
equal : ℕ → ℕ → Bool
equal zero zero = true
equal zero (suc b) = false
equal (suc a) zero = false
equal (suc a) (suc b) = equal a b
```

Agda's mixfix notation can also be used in function names, and this makes it easy to add syntactic constructs. For example, Haskell's `if then else` construct is not supported by Agda out of the box, but one can define such a construct as follows:

```
if_then_else_ : Bool → ℕ → ℕ → ℕ
if_then_else_ true a b = a
if_then_else_ false a b = b
```

Here we pattern match on the first argument, the boolean specified after the `if` term. If it is `true`, we return the argument specified after the `then` term. If it is `false`, we return the argument specified after the `else` term. This can then be used as follows:

```
par : ℕ → ℕ
par n = if equal n (suc zero)
```

```
        then zero
        else suc n
```

If the argument to the function `par` is 1, it returns 0; otherwise it returns the successor of the argument. This could also have been easily implemented by pattern matching on the argument.

However, there is a caveat with our current implementation of the `if then else` construct: the return values from the `then` and `else` branches must be natural numbers. In Haskell, however, the return values can be of any type, as long as the types of the values in both branches agree. To support that, we would need a generic version of our function. It turns out that generics is just a simple application of dependent types.

1.3.3 Dependent types

Consider the functions we've defined up to this point, and, in particular, their type signatures. In all of them we specify a list of types, separated by arrows, that denote the types of the arguments that the function takes as input, except for the last one, which specifies the return type of the function. In the definitions above, as well as in many programming languages, these types are completely fixed: no matter how we call the function, the types of the input arguments, as well as the return type, will remain the same.

With dependent types, however, this will not be true. Dependent types allow the type of an argument to change depending on the *values* given for the preceding arguments. In Agda this is achieved by giving the type for an argument a label, which can then be referenced in later parts of the type signature. When someone calls the function, every time a new argument, with a corresponding type label, is fed to the function, the remaining part of the type signature is updated by replacing all occurrences of the respective label with the new value. The next argument's type, which could have been referencing previous type labels, as well as doing some computation on them, e.g. by calling other functions, will then be evaluated.

This is very different from how type signatures work in most other programming languages, so let's proceed with some examples. Say we have three sets A, B and C. By the distributive law of sets, we know that the sets $A \times (B \sqcup C)$ and $(A \times B) \sqcup (A \times C)$ are the same. Unfortunately this fact does not translate directly to our Agda sets, as the types `A × (B ⊔ C)` and `(A × B) ⊔ (A × C)` are distinct, and their elements are disjoint. We can, however, create a bijection between the elements of the two sets. For now we will represent this as a function taking an element from the left set, and returning an element of the right set. However, we want this function to work for all sets A, B and C. This can be achieved using dependent types as follows:

```
distribute : (A : Set)
           → (B : Set)
           → (C : Set)
           → A × (B ⊔ C)
           → (A × B) ⊔ (A × C)
distribute A B C (a , inj₁ b) = inj₁ (a , b)
distribute A B C (a , inj₂ c) = inj₂ (a , c)
```

Here we take as arguments the three sets A, B and C. After they have been passed to the function, the type of the single input argument, as well as the type of the return value, can be computed. These three arguments are otherwise completely ignored in the body of the function. In fact, we can explicitly ignore them in our function by giving them the special variable name '_':

```
distribute : (A : Set)
           → (B : Set)
           → (C : Set)
           → A × (B ⊔ C)
           → (A × B) ⊔ (A × C)
distribute _ _ _ (a , inj₁ b) = inj₁ (a , b)
distribute _ _ _ (a , inj₂ c) = inj₂ (a , c)
```

The function then does pattern matching on the last input argument, carrying out our bijection. We can call our function as follows:

```
distribute Bool Bool ℕ (true , inj₂ zero)
```

As expected, this will return `inj₂ (true , zero)`.

Although not really necessary in this case, we can simplify the type signature of our function a bit. First, when we want to assign labels to multiple consecutive arguments that have the same type, they can be combined as follows:

```
distribute : (A B C : Set)
           → A × (B ⊔ C)
           → (A × B) ⊔ (A × C)
distribute _ _ _ (a , inj₁ b) = inj₁ (a , b)
distribute _ _ _ (a , inj₂ c) = inj₂ (a , c)
```

Second, as we are using the labels A, B and C as parameters to both the `_×_` and the `_⊔_` data types, Agda can infer that A, B and C must be sets. Hence, instead of explicitly saying that the these three arguments have type `Set`, we can tell Agda to try to infer the types by replacing the type with an underscore.[5]

```
distribute : (A B C : _)
           → A × (B ⊔ C)
           → (A × B) ⊔ (A × C)
distribute _ _ _ (a , inj₁ b) = inj₁ (a , b)
distribute _ _ _ (a , inj₂ c) = inj₂ (a , c)
```

As asking Agda to infer types in this way is such a common pattern, it provides syntactic sugar to make that easier: the `∀` notation. The following definition is equivalent to the previous one:

[5] While an underscore also has special meaning in this case, this is a different meaning from when it is used as a variable name.

```
distribute : ∀ A B C
           → A × (B ⊔ C)
           → (A × B) ⊔ (A × C)
distribute _ _ _ (a , inj₁ b) = inj₁ (a , b)
distribute _ _ _ (a , inj₂ c) = inj₂ (a , c)
```

Let us now have another look at our `if then else` construct from before. We wanted to make our function generic, and now we've seen a way to do that with dependent types. That is, by passing in a type T to our function, we can adjust the remainder of the type signature to ask for a boolean as well as two values of type T, one for each branch, and then give a return value of type T. However, passing this type as an argument to the function every time we want to use the `if then else` construct would be cumbersome, and the usage would be completely different from Haskell's `if then else` construct.

Fortunately Agda provides a solution to this by allowing one to hide certain parameters. Their values will then be inferred by Agda, similar to what was done for the types of A, B and C from before. To hide a parameter, it is surrounded by curly brackets. Our generic `if then else` construct can now be defined as follows:

```
if_then_else_ : {T : Set} → Bool → T → T → T
if_then_else_ true a b = a
if_then_else_ false a b = b
```

It can then be used exactly as before, but now one can specify arguments of arbitrary types in the `then` and `else` branches, as long as the two branches have an argument of the same type.

One thing to note is that the hidden T argument is not listed in the argument list in the two function cases. If we wanted to access its value, which is unnecessary in this case, we could do so as follows:

```
if_then_else_ : {T : Set} → Bool → T → T → T
if_then_else_ {T} true a b = a
if_then_else_ {T} false a b = b
```

We would now be able to pattern match on this argument, or do anything we would otherwise be able to do with a normal argument.

In a similar manner, if we wanted to give the hidden T argument an explicit value when calling the function, we could do so as follows:

```
if_then_else_ {Bool} true zero zero
```

This would not type check, as now the function expects three booleans, but instead is provided with one boolean and then two natural numbers.

Dependent types can also be used in algebraic data types. We've already seen how to specify the set of all natural numbers, but let's now look at the finite set of natural numbers smaller than n. For a fixed n, we could exhaustively list the elements of the set as constructors: `zero`, `one`, `two`, ..., up to $n - 1$, but this would be cumbersome to work with, and won't work for an arbitrary n.

Instead we can make use of dependent types. We will make a data type, `Fin`, which is parameterized by n, where `Fin n` denotes the set of natural numbers smaller than n. First, the number 0 is present in all of these sets where $n \geq 1$. Second, if x is an element of the set of natural numbers smaller than n, then the successor of x is an element of the set of natural numbers smaller than $n + 1$. This can be specified as follows:

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

Agda is no regular programming language, as it is also what is known as a proof assistant. This means that it is possible to specify mathematical proofs in Agda, and their validity will be checked by Agda. Dependent types make this possible, and something known as the Curry-Howard correspondence gives a way to translate mathematical theorems, and their proofs, into code. Intuitively the correspondence works as follows:

- Propositions become types, and proofs become values.
- Implications become functions. That is, their input types give the premises, and the return type is the conclusion. The values given to the function will then be the proofs of the respective premises, and the value returned by the function will be a proof of the conclusion.
- Conjunction becomes our `_×_` type, and disjunction becomes our `_⊔_` type.
- Mathematical induction becomes recursion.
- Case analysis becomes pattern matching.

Let's look at how we can specify and prove theorems in Agda. Consider our `equal` function from before. Currently it isn't very rigorously defined; even if `equal a b` returns `true`, does that really mean that `a` and `b` are equal? For all we know, the `equal` function could have a bug, causing it to return `true` no matter what the inputs are.

What we want is some way to specify that two elements are the same. Here we will be using Agda's definition of equality: two elements are the same if they reduce to the exact same structural definition. For example, both `suc (suc zero)` and `suc zero + suc zero` reduce to `suc (suc zero)`, and are thus the same.

To achieve this we will introduce an algebraic data type `_≡_`, parameterized by two *values* `a` and `b`. The type `a ≡ b` denotes the proposition that `a` and `b` are the same element. So what should its constructors be? Since values are proofs, and constructors provide values, the constructors will actually be providing us with a proof of the given proposition. The constructors therefore form the axioms for our propositions.

Starting off with something simple, we will provide a constructor that only asserts that the proposition `a ≡ a` is true, i.e. that reflexivity holds. Our algebraic data type is then defined as follows:

```
data _≡_ {T : Set} : T → T → Set where
  refl : ∀ {a} → a ≡ a
```

Now say we want to prove that `suc (suc zero) ≡ suc zero + suc zero`. Since Agda will start by reducing the whole expression, this will be equivalent to proving `suc (suc zero) ≡ suc (suc zero)`. But this, of course, follows directly from our `refl` axiom. This proposition can therefore be proved as follows:

```
2=1+1 : suc (suc zero) ≡ suc zero + suc zero
2=1+1 = refl
```

We could also try proving an assertion such as $2 = 1 + 0$ in the same way:

```
2=1+0 : suc (suc zero) ≡ suc zero + zero
2=1+0 = refl
```

Agda will reduce the given expression down to `suc (suc zero) ≡ suc zero`, but since this does not fit the type of `refl`, this code will not type check.

Now, since we have reflexivity as an axiom in our data type, one might wonder if the other two properties of equivalence relations, symmetry and transitivity, should also be axioms. While that would be fine, it turns out that they don't need to be axioms. Instead we can prove that these properties hold for the simple definition that we currently have.

Let's start with symmetry. We want to prove that if `a ≡ b` holds, then `b ≡ a` holds. We will call this property `sym`, and define it as follows:

```
sym : {T : Set} {a b : T} → a ≡ b → b ≡ a
sym p = ?
```

This can be read as follows: if `T` is a set (or a type, if you will), and `a` and `b` are members of that set, and we are given a proof that `a` and `b` are the same, then we will return a proof that `b` and `a` are the same. The function takes only a single (visible) argument, the proof that `a` and `b` are the same, and we will denote that proof with `p`.

In the body of the function we have a question mark. In Agda, this denotes a *hole*. A hole is special placeholder that tells Agda that we have yet to fill in its contents, but would still like Agda to type check the rest of the code. This also allows us to inspect what type Agda expects us to provide in its place. In the case above, it wants us to provide something of type `b ≡ a`.

Now consider the proof `p`, which is of type `a ≡ b`. Let's pattern match on this variable. There is only one constructor for this type, `refl`, so we will only have one case:

```
sym : {T : Set} {a b : T} → a ≡ b → b ≡ a
sym refl = ?
```

At this point, Agda will notice that the first argument actually has type `a' ≡ a'`, because that's the type of `refl`. Furthermore, since it was expecting something of type `a ≡ b`, it will further deduce that `a` and `a'` are actually the same types, as well as that `b` and `a'` are the same types. This, in turn, means that `a` and `b` are the same types. All of this logic is carried out by Agda while it performs type checking.

If we now look at the type that Agda expects in the hole, it turns out to be `a ≡ a`. And of course, this is the type of `refl`, so we can now simply return that value to prove our proposition:

```
sym : {T : Set} {a b : T} → a ≡ b → b ≡ a
sym refl = refl
```

In exactly the same manner we can prove that transitivity holds:

```
trans : {T : Set} {a b c : T} → a ≡ b → b ≡ c → a ≡ c
trans refl refl = refl
```

So now we've proved that our definition of equality forms an equivalence relation. However, it turns out to have an even stronger meaning in Agda: if two values are the same, we can swap one value out for the other one, in any expression, and the new expression will be the same as the first one. That is, our definition of equality also forms a congruence. We can prove this as follows:

```
cong : {S T : Set} → {a b : S} → (f : S → T) → a ≡ b → f a ≡ f b
cong f refl = refl
```

Here `a` and `b` are values of type `S`, and `f` is an arbitrary function that takes in a value of type `S`, and returns a value of type `T`. The proposition then states that if `a` and `b` are the same, then `f` applied to `a` will give the same result as `f` applied to `b`.

With these fundamental properties proved, we can now start to prove some basic propositions. Let's see how we can use them to prove some simple algebraic identities.

Example 10. Let's prove the fact that addition is associative, i.e. that $(a + b) + c = a + (b + c)$ for all natural numbers a, b and c. This proposition can be specified as follows in Agda:

```
+-assoc : ∀ a b c → (a + b) + c ≡ a + (b + c)
```

In order to prove this, we will proceed with induction on a. If $a = 0$, we want to prove that `(zero + b) + c = zero + (b + c)`. By using our definition of _+_, Agda will reduce both of these expressions to `b + c`. And this can be trivially proved by `refl`:

```
+-assoc zero b c = refl
```

If, on the other hand, $a > 0$, then `a` is `suc a'` for some natural number `a'`. In that case we want to prove that `(suc a' + b) + c = suc a' + (b + c)`. Agda will reduce the left expression to `suc ((a' + b) + c)`, and the right expression to `suc (a' + (b + c))`. By induction (recursion) we can prove that `(a' + b) + c ≡ a' + (b + c)`. This, together with the fact that our equality forms a congruence (as proved by the `cong` function), is enough to prove this case: `suc ((a' + b) + c) ≡ suc (a' + (b + c))`.

```
+-assoc (suc a') b c = cong suc (+-assoc a' b c)
```

Our property, and its proof, can now be specified as follows:

```
+-assoc : ∀ a b c → (a + b) + c ≡ a + (b + c)
+-assoc zero b c = refl
+-assoc (suc a') b c = cong suc (+-assoc a' b c)
```

Agda will happily type check this, certifying that the proof is valid.

Example 11. Let's now prove the fact that addition is commutative, i.e. that $a + b = b + a$ for all natural numbers a and b. This can be specified as follows in Agda:

```
+-comm : ∀ a b → a + b ≡ b + a
```

This time let's try doing induction on b. If $b = 0$, then we want to prove that `a + zero ≡ zero + a`. However, Agda will only reduce this to `a + zero ≡ a`. This is because of how we defined addition: we pattern match on the left argument. In this case Agda doesn't know whether the left argument is of the form `zero` or `suc x`, for some `x`, so it can't reduce the expression any further. Doing induction on a would have led to a similar issue.

The fact that `a + zero ≡ a` seems like a useful property in and of itself, so we will create a helper function (a lemma, if you will) to prove it. Its proof is similar to the proof in the previous example:

```
+-right-identity : ∀ a → a + zero ≡ a
+-right-identity zero = refl
+-right-identity (suc a') = cong suc (+-right-identity a')
```

Now we can simply delegate our base case to this helper function:

```
+-comm a zero = +-right-identity a
```

If $b > 0$, `b` is of the form `suc b'`, and we need to prove `a + suc b' ≡ suc (b' + a)`. We will prove this in two steps. First, as a lemma, we will prove that `a + suc b' ≡ suc (a + b')`. After that proving that `suc (a + b') = suc (b' + a)` will be similar to what we've done before: congruence and induction. These two proof steps can then be strung together with our `trans` function, i.e. the fact that our equality is transitive.

Let's start with our lemma:

```
suc-lemma : ∀ a b → a + suc b ≡ suc (a + b)
suc-lemma zero b = refl
suc-lemma (suc a') b = cong suc (suc-lemma a' b)
```

The second case can now be proved as follows:

```
+-comm a (suc b') = trans (suc-lemma a b') (cong suc (+-comm a b'))
```

Our property, and its proof, can now be specified as follows:

```
+-comm : ∀ a b → a + b ≡ b + a
+-comm a zero = +-right-identity a
+-comm a (suc b') = trans (suc-lemma a b') (cong suc (+-comm a b'))
```

As we saw in this example, proofs start to become more complex when they have multiple steps. To make it more natural to specify such proofs, Agda provides something known as "equational reasoning". This is simply a syntax extension, defined using Unicode letters and mixfix notation. To give an example, consider the following alternative proof of commutativity of addition, using equational reasoning:

```
+-comm : ∀ a b → a + b ≡ b + a
+-comm a zero = +-right-identity a
+-comm a (suc b') = begin
  a + suc b'   ≡⟨ suc-lemma a b' ⟩
  suc (a + b') ≡⟨ cong suc (+-comm a b') ⟩
  suc (b' + a) ∎
```

To begin a proof in the equational reasoning environment, we use the `begin` identifier. On the left side we specify a sequence of expressions. On the right side, within brackets, we give a proof that the expression to the left is the same as the expression on the line below. Behind the scenes, the environment then uses transitivity, just as we did in our previous proof, to get the complete proof.

When developing proofs using the equational reasoning environment, it can be very handy to insert a hole, i.e. a question mark, in one or more of the brackets on the right. This allows one to develop the proof step by step. The expressions on the left then serve two purposes; to help inform Agda's type checker what kind of proof is expected within the pairs of brackets, as well as for us to keep our sanity.

So far we have seen a few proofs, all of which show that the given proposition is true. However, we can also show that some propositions are false. Recall that a proof of a property, represented as a type, is simply any element of that type. Hence, if a proposition is false, it means that there are no elements of the respective type. In particular, consider the following empty algebraic data type, often known as *bottom*:

```
data ⊥ : Set where
```

The idea, then, is that if we have a proposition `p`, and from `p` we can deduce `⊥`, then `p` must be false. Or, from an alternative view point, if we are given an element of type `p`, and from it we can construct an element of type `⊥`, that is a contradiction, meaning that we never had an element of `p` to begin with.

Agda also provides one more feature to help with contradictions. Say we are implementing a function, and we are given an argument of type `p`. If Agda can deduce that the type `p` is empty, meaning that it would actually never be possible to provide this particular argument to the function, we can opt out of implementing the remainder of this case by pattern matching the current argument with the special identifier `()`, representing a contradiction.

With this machinery in hand, we can prove that $0 \neq 1$ as follows:

```
0≢1 : zero ≡ suc zero → ⊥
0≢1 ()
```

This type checks, meaning that if we ever have a proof of `zero ≡ suc zero`, we can also prove bottom, a contradiction.

This pattern of proving that a proposition `p` implies bottom is quite common, and so Agda provides syntax to make it more natural to specify. The previous definition is equivalent to the following.

```
0≢1 : ¬ (0 ≡ 1)
0≢1 ()
```

There is one thing worth mentioning here, which is that Agda's logic is only sound if all functions eventually terminate. If this were not the case, we could prove whatever we want:

```
contr : ⊥
contr = contr
```

This function passes type checking, but does not pass Agda's *termination checker*, which makes sure all functions terminate. Of course, deciding whether a given program terminates is well known to be an impossible task [34]. Instead, Agda's termination checker looks for some common programming patterns that guarantee that the function terminates. If it is unable to do so, it will simply fail, asking the programmer to implement the function in a different way. When that happens, finding a way to implement the given task in a way that makes the termination checker happy can be pretty difficult. Luckily this doesn't happen too often.

Recall our `equal` function from before: given two natural numbers a and b, it returned `true` if they were the same, and `false` otherwise. As we mentioned above, this isn't a very rigorous definition. Ideally we would get a proof of the result: if they are equal, we would get something of type `a ≡ b`; otherwise we would get something of type `a ≡ b → ⊥`.

More generally, if we have a property p, we sometimes want to decide if it's true or false. Agda provides a data type to represent the result from such a decision procedure:

```
data Dec (p : Set) : Set where
  yes : p → Dec p
  no : (p → ⊥) → Dec p
```

Before concluding our discussion about dependent types, let us introduce one more data type. So far we've talked quite a bit about bijections, and they play a crucial role in our work. We have also already given an example of a bijection in Agda, namely our `distribute` function. There are two issues with this definition, however. First, it is not possible to get the inverse of the bijection, which is often useful. And second, we don't know whether it actually forms a bijection.

To fix the first issue, we can simply keep track of two functions: the original function f and its inverse $g = f^{-1}$. This, however, leads to a third issue: we don't know if f

and g are actually each other's inverses. This can be fixed by proving two properties: $f(g(x)) = x$ and $g(f(x)) = x$, for all x. In fact, this is also enough to prove that both f and g are bijections: just the fact that they are each other's inverses shows that they form a one-to-one correspondence between the two domains.

We can now represent a bijection between two sets A and B with the following algebraic data type:

```
data _≅_ (A B : Set) : Set where
  mkBij : (to : A → B)
        → (from : B → A)
        → (∀ b → to (from b) ≡ b)
        → (∀ a → from (to a) ≡ a)
        → A ≅ B
```

The `distribute` function can now be specified more rigorously as follows:

```
distribute' : {A B C : Set} → (A × (B ⊔ C)) ≅ ((A × B) ⊔ (A × C))
distribute' {A} {B} {C} = mkBij to from to-from from-to
  where
    to : A × (B ⊔ C) → (A × B) ⊔ (A × C)
    to (a , inj₁ b) = inj₁ (a , b)
    to (a , inj₂ c) = inj₂ (a , c)

    from : (A × B) ⊔ (A × C) → A × (B ⊔ C)
    from (inj₁ (a , b)) = (a , inj₁ b)
    from (inj₂ (a , c)) = (a , inj₂ c)

    to-from : ∀ b → to (from b) ≡ b
    to-from (inj₁ (a , b)) = refl
    to-from (inj₂ (a , c)) = refl

    from-to : ∀ a → from (to a) ≡ a
    from-to (a , inj₁ b) = refl
    from-to (a , inj₂ c) = refl
```

Here the `where` notation is used to define functions in a local scope.

Finally, as we did with propositional equality, we can prove that bijections form an equivalence relation:

```
Brefl : ∀ {A} → A ≅ A
Brefl {A} = mkBij to from to-from from-to
  where
    to : A → A
    to a = a

    from : A → A
    from a = a
```

```
    to-from : ∀ a → to (from a) ≡ a
    to-from y = refl

    from-to : ∀ a → from (to a) ≡ a
    from-to a = refl

Bsym : ∀ {A B} → A ≅ B → B ≅ A
Bsym (mkBij A→B B→A to-from from-to) = mkBij B→A A→B from-to to-from

Btrans : ∀ {A B C} → A ≅ B → B ≅ C → A ≅ C
Btrans {A} {B} {C} (mkBij A→B B→A to-from₁ from-to₁)
                   (mkBij B→C C→B to-from₂ from-to₂)
  = mkBij to from to-from from-to
  where
    to : A → C
    to a = B→C (A→B a)

    from : C → A
    from c = B→A (C→B c)

    to-from : ∀ c → to (from c) ≡ c
    to-from c = trans (cong B→C (to-from₁ (C→B c))) (to-from₂ c)

    from-to : ∀ a → from (to a) ≡ a
    from-to a = trans (cong B→A (from-to₂ (A→B a))) (from-to₁ a)
```

1.3.4 Standard library

Agda's standard library already ships with a host of commonly used functions and data types. Here are some of the ones we will be using:

Data.Nat includes the natural number data type, ℕ, presented above, as well as operations such as addition and multiplication.

Data.Nat.Properties.Simple includes proofs of many simple algebraic identities. We will abbreviate this path as `NPS`, as it will come in handy later.

Data.Fin includes the finite sets, `Fin`, presented above.

Data.Maybe includes a data type `Maybe`, which only has two constructors; `nothing` and `just x`. Often used as the return value of a function that may or may not return something useful.

Data.List includes the `List` data type, which represents an ordered sequence of elements of a given type. The list type has two constructors; `[]` represents the empty list, and `x ∷ xs` represents prepending the element `x` to the list `xs`. It also includes some of the standard functions (from functional programming) on lists, such as `map`.

Data.Vec includes the `Vec` data type, which is just a list of a fixed size.

1.3.5 Syntax

Let's go over some of Agda's syntax that we will be using, but haven't seen so far.

Despite our extensive use of the unary representation for natural numbers, Agda allows one to use literals in base 10: 0, 1, 2, ... These will simply be expanded to their unary representation at compile time.

Agda also supports much of Haskell's syntax, perhaps slightly altered. We've already seen the `where` notation to specify functions within a local scope. Similarly Agda supports the `let in` notation, which allows one to define variables within an expression: `let x = 5 in x + 1` will reduce to `6`.

Agda also allows one to specify anonymous, so-called lambda functions:

```
let f = λ x → x + 1 in f 5
```

This will also reduce to `6`. However, unlike Haskell, Agda allows one to do pattern matching within an anonymous function. This is achieved by specifying the pattern matching within curly brackets, as follows:

```
λ { zero → zero ; (suc x) → x }
```

This anonymous function decreases the input argument by one, unless the argument is 0, in which case it returns 0.

Finally, Agda also supports something known as `with` notation. When implementing a function, it is very often useful to do pattern matching on its arguments. The results of the matching may then be processed further. However, after processing, one may want to do a second round of pattern matching based on the results. Before the `with` notation was introduced, one would have to define a second function to perform the pattern matching. However, using `with` notation this becomes as easy as follows:

```
fun : ℕ → ℕ
fun zero = zero
fun (suc x) with process₁ x | process₂ x
fun (suc x) | zero | zero = zero
fun (suc x) | zero | (suc x₂) = x₂
fun (suc x) | (suc x₁) | x₂ = x₁ + x₂
```

In this example, which is completely fabricated, the `with` notation is used with two arguments, but it can be used with one or more arguments. It is also possible to nest applications of the `with` notation. Sometimes it may become cumbersome to repeat the first part of the function definition for each of the subsequent cases; this can be skipped by simply typing `...` instead:

```
fun : ℕ → ℕ
fun zero = zero
fun (suc x) with process₁ x | process₂ x
... | zero | zero = zero
... | zero | (suc x₂) = x₂
... | (suc x₁) | x₂ = x₁ + x₂
```

This concludes our short introduction to Agda. Unfortunately we only got to scrape the surface of Agda and dependent types; we refer the interested reader to [21, 23, 33] to learn more about them, as well as their many interesting applications.

Chapter 2

The translate module

Here we finally come to our main contribution. As we pointed out at the end of Section 1.2, the implementation of the translation method given by Wood and Zeilberger [37] has some shortcomings. Now that we've given a quick introduction to Agda, we can describe our implementation of the translation method, which addresses the shortcomings of the previous implementation, as well as provides some additional features. Our full implementation can be found on GitHub [16]. See also Appendix A for instructions on how to set up the module.

As we've seen, it's possible to give algebraic proofs for identities in Agda. The underlying idea of our implementation is to build on top of that, and make it easy, and hopefully even seamless, to lift such a proof to a bijection. That is, however, not immediately possible. When given something of type `a ≡ b`, we have no way to access the individual proof steps, which is necessary when applying the translation method.

To work around that, we re-implemented many of the data types and functions that are commonly used when doing algebraic proofs in Agda, keeping track of all information needed to apply the translation method. The data types and functions include natural numbers, operations on natural numbers, propositional equality, and various basic algebraic identities. When implementing these we tried to be consistent with Agda's implementations, in particular using the same names when possible. Agda makes this very easy, as all of these are implemented in the standard library, but not built-in as in many other languages (natural numbers in particular).

With the new versions of the built-in data types and functions, it should be possible to implement an algebraic proof of an identity in Agda in almost the same way as previously, but now with the ability to call a single function to apply the translation method and get the lifted bijection.

Our implementation still has some caveats, however. Some functionality, such as the `rewrite` built-in[1] and the general `cong` function, has not been implemented, and that may not even be possible without altering Agda's core functionality. Sometimes it may also be necessary to provide explicit bijections, especially when making new combinatorial definitions or when providing base cases for induction. This is inherent to the translation method, and the only thing we can do about that is make defining such bijections as simple as possible.

[1] Given a proof that `a ≡ b`, `rewrite` allows one to replace all occurrences of `a` in a type with `b`. It can be handy in some proofs, but is really just syntactic sugar for a `with` statement with some pattern matching.

One final caveat is that it is not possible to do the same kind of pattern matching on our natural numbers as it is with Agda's. That is, `zero` and `suc x` will not be the only cases. This makes it hard to do some parts of a proof using our version of natural numbers, but in most cases these parts are not being lifted by the translation method. An example are the indices used for recurrences, where pattern matching is important. But as these indices will not be lifted by the translation method, one can simply use Agda's natural numbers.

But this last caveat has some unfortunate consequences. It means that it must be possible for our version and Agda's version of natural numbers to coexist, and that introduces naming conflicts. Since our implementation—and hopefully users of our implementation—will mostly be using our version of the natural numbers, we decided to rename the built-in data types and functions. Here is what we changed, and this will be our notation from here on:

- Data types and functions related to the built-in natural numbers will be prefixed with `ℕ`: `zero` and `suc` become `ℕzero` and `ℕsuc`, respectively, `+` and `*` become `ℕ+` and `ℕ*`, respectively, and so on.
- Data types and functions related to the built-in propositional equality will be prefixed with `P`: `_≡_` becomes `_P≡_`, `refl`, `sym`, `trans` and `cong` become `Prefl`, `Psym`, `Ptrans` and `Pcong`, respectively, and so on.
- Data types and functions related to bijections will be prefixed with `B`: `refl`, `sym` and `trans` become `Brefl`, `Bsym` and `Btrans`, respectively. An exception is the bijection type itself, which is still `_≅_`.

So, with a high-level description of our implementation, its caveats, and this new notation, we can go on to describe each component of our implementation in detail, starting with expressions.

2.1 Expressions

To be able to use the translation method, we need to be able to lift both expressions and identities. In particular, lifting expressions allows us to know which sets we're looking for a bijection between.

So how do we implement lifting of expressions in Agda? Here we have the issue described in the previous section. Namely, when we have an expression such as $a + b$ in Agda, where a and b are natural numbers, all we have access to is the result of the addition, which is just another natural number. However, we need access to the expression tree itself, so that we can, say, use the rule from Table 1.1 to lift $a + b$ to the set $A \sqcup B$, where A and B have been lifted recursively.

Agda does support reflection [8, 36], which allows one to inspect Agda's internal abstract syntax tree for a given expression, so that is certainly something to consider. However, that turns out to be an impractical approach for two reasons. First, when one reflects a given variable x, the resulting syntax tree does not describe how the variable x was computed; instead the syntax tree is simply one node representing the fact that x is a variable. And even if that were the case, the expression x may be the result of a series of non-trivial computations, so one would somehow need to partially

evaluate the abstract syntax trees of these computations to access the required parts that need to be lifted. For these reasons we did not pursue this approach further.

The approach that we did take was to make a data type, mimicking natural numbers, that keeps track of how the natural number was built—at least the parts that are relevant for lifting. This data type is called `Expr`, as it represents expressions, and a simplified version of it is as follows.

```
data Expr : Set where
  zero : Expr
  suc : (x : Expr) → Expr

  _+_ : (l r : Expr) → Expr
  _*_ : (l r : Expr) → Expr

  2^ : (n : ℕ) → Expr
  fib : (n : ℕ) → Expr
```

The actual implementation contains more constructors for other commonly used functions, and we will see some of these later.

With this data type, and more specifically its constructors, one can now type up an Agda expression such as `suc (suc zero) * fib 4 + 2^ 3`, which is common when doing algebraic proofs, but then go on to inspect the expression and see how it was built. The caveat here, though, as mentioned before, is that these expressions cannot be treated exactly as natural numbers. In particular, if one pattern matches on such an expression, it is not sufficient to consider the cases `zero` and `suc x`, but also each of the numerous other constructors defined for the `Expr` data type.

Sometimes it will nevertheless be necessary to get the actual value of the expression as a natural number. To allow for that we provide the `value` function, defined as follows:

```
value : Expr → ℕ
value zero = ℕzero
value (suc x) = ℕsuc (value x)
value (l + r) = value l ℕ+ value r
value (l * r) = value l ℕ* value r
value (2^ n) = ℕ2^ n
value (fib n) = ℕfib n
```

Here we use the usual natural number operations prefixed with ℕ, as well as the following two simple recurrences:

```
ℕ2^ : ℕ → ℕ
ℕ2^ 0 = 1
ℕ2^ (ℕsuc n) = 2 ℕ* ℕ2^ n

ℕfib : ℕ → ℕ
ℕfib 0 = 1
ℕfib 1 = 1
ℕfib (ℕsuc (ℕsuc n)) = ℕfib (ℕsuc n) ℕ+ ℕfib n
```

Similar to the `value` function, we can now define the `lift` function, which takes an expression and lifts it to a set—exactly what we were looking to be able to do. Using Table 1.1, this is implemented as follows:

```
lift : Expr → Set
lift zero = ⊥
lift (suc x) = Maybe (lift x)
lift (l + r) = lift l ⊔ lift r
lift (l * r) = lift l × lift r
lift (2^ n) = BinStr n
lift (fib n) = FibStr n
```

As we saw in our introduction to Agda, the data types on the right side can be considered as the sets of their inhabitants. In particular:

- $\bot$ is the empty type and has no inhabitants, and can thus be considered as the empty set. Its cardinality is 0, so 0 can be lifted to it.
- `Maybe A` is either the special value `nothing`, or `just a`, where `a` is any element of `A`. Hence it can be thought of as the set $A \sqcup \{\texttt{nothing}\}$. It has cardinality $1 + |A|$, which is why we used it in the case `suc x`.
- $\sqcup$ and $\times$ can be thought of as the disjoint union and Cartesian product, respectively.

For the last two cases we defined the following algebraic data types:

```
data BinStr : (n : ℕ) → Set where
  [] : BinStr ℕzero
  _∷_ : ∀ {n} → Fin 2 → BinStr n → BinStr (ℕsuc n)

data FibStr : (n : ℕ) → Set where
  [] : FibStr ℕzero
  _∷1 : ∀ {n} → FibStr n → FibStr (ℕsuc n)
  _∷2 : ∀ {n} → FibStr n → FibStr (ℕsuc (ℕsuc n))
```

The first one represents binary strings of length n, and the second one represents integer compositions of n using only the integers 1 and 2. They have 2^n and F_n inhabitants, as required.

Recall that it is possible to lift an expression in many different ways, and hence the above lifted variants are just one of many possibilities. Although unlikely, if one requires that one of the operations be lifted in another way, this can be worked around in couple of ways:

- Redefine the above implementation of `lift` to make it work as required.
- Add a new constructor to the `Expr` type, perhaps with a similar but distinct name, that is then lifted as expected.

- Provide a bijection between the above lifted type and the required lifted type. This can then be used as shown in Section 2.2 to use the two distinct lifted variants interchangeably.

Now, to make it easier to work with expressions, as well as to do interesting things with them, we have also defined various helper functions.

First, consider the natural number 5. To represent it as an expression, one needs to type `suc (suc (suc (suc (suc zero))))`. This becomes cumbersome when done frequently. We considered changing the meaning of literals in Agda so that one could simply type `5` to get the required expression, but as of version 2.4.0 of Agda 2 this is no longer possible for data types that have constructors other than `zero` and `suc` (see [24]). Instead we defined the following function:

```
nat : ℕ → Expr
nat ℕzero = zero
nat (ℕsuc n) = suc (nat n)
```

So now one can type `nat 5`—not as simple as `5`, but simpler than using `suc` and `zero` explicitly.

Then we have three functions which will be useful later when displaying and verifying bijections.

- The function `show`, which takes in an expression, and then an element of its lifted type, and provides a textual representation of the element. Its type signature is:

  ```
  show : (E : Expr) → (x : lift E) → String
  ```

- The function `equal`, which takes in an expression, and then two elements of its lifted type, and decides whether the two elements are structurally equivalent. Its type signature is:

  ```
  equal : (E : Expr) → (x y : lift E) → Dec (x P≡ y)
  ```

- The function `generate`, which takes in an expression, and then returns a list of all the elements of its lifted type. Its type signature is:

  ```
  generate : (E : Expr) → List (lift E)
  ```

It would be a bit much to show the full implementation, but here follows the particular case where the lifted type is `FibStr n`:

```
show : ∀ {n} → FibStr n → String
show [] = "[]"
show (x ::1) = show x ++ " ::1"
show (x ::2) = show x ++ " ::2"

equal : ∀ {n} → (x y : FibStr n) → Dec (x P≡ y)
equal [] [] = yes Prefl
equal (x ::1) (y ::1) with equal x y
```

```
equal (x ::1) (y ::1) | yes Prefl = yes Prefl
equal (x ::1) (y ::1) | no ¬p = no (λ { Prefl → ¬p Prefl })
equal (x ::2) (y ::2) with equal x y
equal (x ::2) (y ::2) | yes Prefl = yes Prefl
equal (x ::2) (y ::2) | no ¬p = no (λ { Prefl → ¬p Prefl })
equal (x ::1) (y ::2) = no (λ ())
equal (x ::2) (y ::1) = no (λ ())

generate : ∀ {n} → List (FibStr n)
generate {ℕzero} = [] ∷ []
generate {ℕsuc ℕzero} = ([] ::1) ∷ []
generate {ℕsuc (ℕsuc n)}
  = map _::1 (generate {ℕsuc n}) ++ map _::2 (generate {n})
```

Notice that the type for the `generate` function does not inspire much confidence in its implementation—for all we know it could be returning the empty list, no matter what the expression is. Later we will require a proof that this does indeed generate all the elements of the respective type. In particular, in Section 2.7 we will need to prove that the sets we're working with are finite, and this will make that possible.

So how do we prove that the list contains all elements of the respective type? Concretely, for any element of that type, we want to prove that the given element is in the list that `generate` returns. We named this property `exhaustive`, and it has the following type definition[2]:

```
exhaustive : (E : Expr) → (e : lift E) → e ∈ generate E
```

Again, it would be too much to show the full implementation, so here we show the particular case where the lifted type is `FibStr n`:

```
exhaustive : ∀ {n} x → x ∈ (generate {n})
exhaustive {ℕzero} [] = here Prefl
exhaustive {ℕsuc ℕzero} ([] ::1) = here Prefl
exhaustive {ℕsuc (ℕsuc n)} (x ::1)
  = ∈++ˡ {_} {map _::1 (generate {ℕsuc n})}
             {map _::2 (generate {n})}
             (∈map (exhaustive x))
exhaustive {ℕsuc (ℕsuc n)} (x ::2)
  = ∈++ʳ {_} {map _::1 (generate {ℕsuc n})}
             {map _::2 (generate {n})}
             (∈map (exhaustive x))
```

2.2 Equivalence of expressions

In the last section we mentioned that, to be able to use the translation method, we need to be able to lift both expressions and identities. Now that we know how to represent and lift expressions, we can consider the identities.

[2]The type `x ∈ xs` represents a proof that `x` is an element of the list `xs`.

So how are identities represented in Agda? As we have seen, they are of the form `a ≡ b`, where `a` and `b` are expressions, and `≡` is propositional equality. When using the translation method, we need to be able to lift these identities to bijections between the respective lifted sets. As there are multiple valid bijections, we will leave it up to whoever proved the identity to provide a suitable natural bijection.

Our new augmented `≡` relation will then carry this bijection between the respective lifted sets, as well as the original propositional equality. It is defined as follows:

```
data _≡_ (a b : Expr) : Set₂ where
  proof : value a P≡ value b
        → lift a ≅ lift b
        → a ≡ b
```

This can be read as: if the values of `a` and `b` are propositionally equivalent, and there is a bijection between their lifted sets, then the expressions `a` and `b` are equivalent.

As expected, this forms an equivalence relation, as proved by the following three properties (`refl` for reflexivity, `sym` for symmetry, and `trans` for transitivity):

```
refl : ∀ {a} → a ≡ a
refl = proof Prefl Brefl

sym : ∀ {a b} → a ≡ b → b ≡ a
sym (proof a=b a≅b) = proof (Psym a=b) (Bsym a≅b)

trans : ∀ {a b c} → a ≡ b → b ≡ c → a ≡ c
trans (proof a=b a≅b) (proof b=c b≅c)
    = proof (Ptrans a=b b=c) (Btrans a≅b b≅c)
```

We can even convince Agda that this is an equivalence relation, as follows:

```
≡-equivalence : IsEquivalence _≡_
≡-equivalence = record
  { refl = refl
  ; sym = sym
  ; trans = trans
  }
```

With this definition of an identity, it now becomes trivial to find the lifted bijection, as all the work has been delegated elsewhere:

```
bijection : ∀ {a b} → a ≡ b → lift a ≅ lift b
bijection (proof _ bij) = bij
```

Similarly, we can also get the original underlying propositional equality, if needed:

```
equality : ∀ {a b} → a ≡ b → value a P≡ value b
equality (proof prf _) = prf
```

With these definitions, we can now start to prove identities. Let's begin with some simple arithmetic identities that we've seen before.

Example 12. The identity $a + b = b + a$ represents the commutativity of addition, and we saw how to prove it in Agda in Example 11. Furthermore we saw how to lift this identity to a bijection in Example 3. Combining these two examples, we can now prove that the identity holds under our new definition of equivalence:

```
+-comm : ∀ {a b} → a + b ≡ b + a
+-comm {a} {b} = proof (NPS.+-comm (value a) (value b))
                       (mkBij to from to-from from-to)
  where
    to : lift (a + b) → lift (b + a)
    to (inj₁ x) = inj₂ x
    to (inj₂ x) = inj₁ x

    from : lift (b + a) → lift (a + b)
    from (inj₁ y) = inj₂ y
    from (inj₂ y) = inj₁ y

    to-from : ∀ y → to (from y) P≡ y
    to-from (inj₁ x) = Prefl
    to-from (inj₂ y) = Prefl

    from-to : ∀ x → from (to x) P≡ x
    from-to (inj₁ x) = Prefl
    from-to (inj₂ y) = Prefl
```

Example 13. The identity $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ represents the associativity of multiplication, and it has already been proved in Agda's standard library as `NPS.*-assoc`. Furthermore we saw how to lift this identity to a bijection in Example 4. We can now prove that the identity holds under our new definition of equivalence:

```
*-assoc : ∀ {a b c} → (a * b) * c ≡ a * (b * c)
*-assoc {a} {b} {c} = proof (NPS.*-assoc (value a) (value b) (value c))
                            (mkBij to from to-from from-to)
  where
    to : lift ((a * b) * c) → lift (a * (b * c))
    to ((x , y) , z) = x , (y , z)

    from : lift (a * (b * c)) → lift ((a * b) * c)
    from (x , (y , z)) = (x , y) , z

    to-from : ∀ y → to (from y) P≡ y
    to-from (x , (y , z)) = Prefl

    from-to : ∀ x → from (to x) P≡ x
    from-to ((x , y) , z) = Prefl
```

Example 14. The identity $a \cdot (b + c) = a \cdot b + a \cdot c$ represents the distributivity of multiplication over addition, and it has already been proved in Agda's standard library as `NP.distribˡ-*-+`. Furthermore we saw how to lift this identity to a bijection in Example 5. Combining these two examples, we can now prove that the identity holds under our new definition of equivalence:

```
distribˡ-*-+ : ∀ {a b c} → a * (b + c) ≡ a * b + a * c
distribˡ-*-+ {a} {b} {c} = proof (NP.distribˡ-*-+ (value a) (value b) (value c))
                                 (mkBij to from to-from from-to)
  where
    to : lift (a * (b + c)) → lift (a * b + a * c)
    to (l , (inj₁ r)) = inj₁ (l , r)
    to (l , (inj₂ r)) = inj₂ (l , r)

    from : lift (a * b + a * c) → lift (a * (b + c))
    from (inj₁ (l , r)) = l , (inj₁ r)
    from (inj₂ (l , r)) = l , (inj₂ r)

    to-from : ∀ y → to (from y) P≡ y
    to-from (inj₁ (l , r)) = Prefl
    to-from (inj₂ (l , r)) = Prefl

    from-to : ∀ x → from (to x) P≡ x
    from-to (l , inj₁ x) = Prefl
    from-to (l , inj₂ y) = Prefl
```

Example 15. Both addition and multiplication form congruences. The fact that addition forms a congruence can be stated as follows: If $a = c$ and $b = d$, then $a + b = c + d$. We can prove that this identity holds under our new definition of equivalence:

```
+-cong : ∀ {a b c d} → a ≡ b → c ≡ d → a + c ≡ b + d
+-cong {a} {b} {c} {d} a≡b c≡d
  = lemma (equality a≡b) (bijection a≡b)
          (equality c≡d) (bijection c≡d)
  where
    lemma : value a P≡ value b
          → lift a ≅ lift b
          → value c P≡ value d
          → lift c ≅ lift d
          → a + c ≡ b + d
    lemma a≡b (mkBij a→b b→a to-from₁ from-to₁)
          c≡d (mkBij c→d d→c to-from₂ from-to₂)
        = proof prf (mkBij to from to-from from-to)
      where
        prf : value (a + c) P≡ value (b + d)
        prf = Ptrans (Pcong (λ y → y ℕ+ value c) a≡b)
                     (Pcong (λ y → value b ℕ+ y) c≡d)
```

```
    to : lift (a + c) → lift (b + d)
    to (inj₁ x) = inj₁ (a→b x)
    to (inj₂ x) = inj₂ (c→d x)

    from : lift (b + d) → lift (a + c)
    from (inj₁ x) = inj₁ (b→a x)
    from (inj₂ x) = inj₂ (d→c x)

    to-from : ∀ y → to (from y) P≡ y
    to-from (inj₁ x) = Pcong inj₁ (to-from₁ x)
    to-from (inj₂ y) = Pcong inj₂ (to-from₂ y)

    from-to : ∀ x → from (to x) P≡ x
    from-to (inj₁ x) = Pcong inj₁ (from-to₁ x)
    from-to (inj₂ y) = Pcong inj₂ (from-to₂ y)
```

Example 16. In the ring of integers 0 is the additive identity. This can be formulated as the identity $a + 0 = a$, and we saw how to prove it in Agda in Example 11. We can prove that this identity holds under our new definition of equivalence:

```
+-right-identity : ∀ {a} → a + zero ≡ a
+-right-identity {a} = proof (NPS.+-right-identity (value a))
                             (mkBij to from to-from from-to)
  where
    to : lift (a + zero) → lift a
    to (inj₁ x) = x
    to (inj₂ ())

    from : lift a → lift (a + zero)
    from x = inj₁ x

    to-from : ∀ y → to (from y) P≡ y
    to-from y = Prefl

    from-to : ∀ x → from (to x) P≡ x
    from-to (inj₁ x) = Prefl
    from-to (inj₂ ())
```

Proceeding in a very similar manner we proved a host of trivial arithmetic identities, including all of those presented in Table 1.2. In fact, with these identities we can prove that the set `Expr`, equipped with the operations `+` and `*`, forms a commutative semiring under our new definition of equivalence:

```
≡-commutativeSemiring : CommutativeSemiring _ _
≡-commutativeSemiring = record
  { Carrier = Expr
  ; _≈_ = _≡_
  ; _+_ = _+_
  ; _*_ = _*_
```

```
  ; 0# = zero
  ; 1# = suc zero
  ; isCommutativeSemiring = record
    { +-isCommutativeMonoid = record
      { isSemigroup = record
        { isEquivalence = ≡-equivalence
        ; assoc = λ x y z → +-assoc {x} {y} {z}
        ; •-cong = +-cong
        }
      ; identityˡ = λ x → trans (+-comm {zero} {x}) (+-right-identity {x})
      ; comm = λ x y → +-comm {x} {y}
      }
    ; *-isCommutativeMonoid = record
      { isSemigroup = record
        { isEquivalence = ≡-equivalence
        ; assoc = λ x y z → *-assoc {x} {y} {z}
        ; •-cong = *-cong
        }
      ; identityˡ = λ x → trans (*-comm {suc zero} {x}) (*-right-identity {x})
      ; comm = λ x y → *-comm {x} {y}
      }
    ; distribʳ = λ x y z → distribʳ-*-+ {x} {y} {z}
    ; zeroˡ = λ x → trans (*-comm {zero} {x}) (*-right-zero {x})
    }
  }
```

While this is interesting in its own right, we will investigate some very interesting uses of this fact in Section 2.5.

We also proved some more complex identities, such as those given by the defining equations of some of the combinatorial objects we've been working with. Consider the following examples.

Example 17. By the definition of exponentiation we have that $2^n = 2 \cdot 2^{n-1}$, when $n \geq 1$. In Agda this is better formulated as $2^{n+1} = 2 \cdot 2^n$, when $n \geq 0$, as it is usually better to avoid subtraction when working with natural numbers. We can prove that this identity holds under our new definition of equivalence:

```
2^-def : ∀ {n} → 2^ (ℕsuc n) ≡ nat 2 * 2^ n
2^-def {n} = proof Prefl (mkBij to from to-from from-to)
  where
    to : lift (2^ (ℕsuc n)) → lift (nat 2 * 2^ n)
    to (Fzero ∷ xs) = nothing , xs
    to (Fsuc Fzero ∷ xs) = just nothing , xs
    to (Fsuc (Fsuc ()) ∷ xs)

    from : lift (nat 2 * 2^ n) → lift (2^ (ℕsuc n))
    from (nothing , xs) = Fzero ∷ xs
    from (just nothing , xs) = Fsuc Fzero ∷ xs
    from (just (just ()) , xs)
```

```
    to-from : ∀ y → to (from y) P≡ y
    to-from (just (just ()) , xs)
    to-from (just nothing , xs) = Prefl
    to-from (nothing , xs) = Prefl

    from-to : ∀ x → from (to x) P≡ x
    from-to (Fzero ∷ xs) = Prefl
    from-to (Fsuc Fzero ∷ xs) = Prefl
    from-to (Fsuc (Fsuc ()) ∷ xs)
```

Example 18. We have the identity $F_n = F_{n-1} + F_{n-2}$, when $n \geq 2$, by the definition of the Fibonacci numbers. As in the previous example, this is better stated as $F_{n+2} = F_{n+1} + F_n$, when $n \geq 0$, to avoid subtraction. We can prove that this identity holds under our new definition of equivalence:

```
fib-def : ∀ {n} → fib (ℕsuc (ℕsuc n)) ≡ fib (ℕsuc n) + fib n
fib-def {n} = proof Prefl (mkBij to from to-from from-to)
  where
    to : lift (fib (ℕsuc (ℕsuc n))) → lift (fib (ℕsuc n) + fib n)
    to (xs ::1) = inj₁ xs
    to (xs ::2) = inj₂ xs

    from : lift (fib (ℕsuc n) + fib n) → lift (fib (ℕsuc (ℕsuc n)))
    from (inj₁ xs) = xs ::1
    from (inj₂ xs) = xs ::2

    to-from : ∀ y → to (from y) P≡ y
    to-from (inj₁ x) = Prefl
    to-from (inj₂ y) = Prefl

    from-to : ∀ x → from (to x) P≡ x
    from-to (x ::1) = Prefl
    from-to (x ::2) = Prefl
```

With all of these identities proved, we now have enough in our toolbox to start combining them to prove more complex identities. Before giving the first non-trivial example, let us introduce some simple tools that proved to be helpful when using the translation method.

2.3 Tools

As we saw in Section 1.3, algebraic proofs can become quite tedious in Agda when they involve more than a couple of steps. As the proofs look basically the same, the same will hold for proofs using our translate module.

For example, consider the identity $a \cdot (b + c) = c \cdot a + a \cdot b$. This can be easily proved algebraically as follows:

$$
\begin{aligned}
a \cdot (b + c) &= a \cdot b + a \cdot c && \text{distributivity} \\
&= a \cdot c + a \cdot b && \text{commutativity} \\
&= c \cdot a + a \cdot b && \text{congruence and commutativity}
\end{aligned}
$$

Using our translate module, this can then be converted to an Agda proof as follows:

```
ex : ∀ {a b c} → a * (b + c) ≡ c * a + a * b
ex {a} {b} {c} = trans distribˡ-*-+ (trans +-comm (+-cong *-comm refl))
```

This example clearly shows how Agda proofs involving just a couple of steps can be much less readable, and harder to work with, than their textual counterparts.

To combat this, we introduced Agda's equational reasoning in Section 1.3. Luckily, Agda's standard library makes it trivial to bring this syntax to other equivalence relations, as follows:

```
-- Prove that Expr coupled with the ≡ relation forms a setoid
≡-setoid : Setoid _ _
≡-setoid = record
  { Carrier = Expr
  ; _≈_ = _≡_
  ; isEquivalence = ≡-equivalence
  }

open import Relation.Binary.EqReasoning ≡-setoid public
  renaming ( _≈⟨_⟩_ to _≡⟨_⟩_
           ; _≡⟨_⟩_ to _P≡⟨_⟩_
           ; _≡⟨⟩_ to _P≡⟨⟩_
           )
```

Now the proofs become much more readable. As an example, consider the proof above, which can now be converted to Agda code as follows:

```
ex2 : ∀ {a b c} → a * (b + c) ≡ c * a + a * b
ex2 {a} {b} {c} = begin
    a * (b + c)   ≡⟨ distribˡ-*-+ ⟩
    a * b + a * c ≡⟨ +-comm ⟩
    a * c + a * b ≡⟨ +-cong *-comm refl ⟩
    c * a + a * b ∎
```

Equational reasoning also makes it much easier to develop the proofs, as holes can be temporarily inserted in the middle of the proof, while sketching the proof steps with the types on the left.

Now, we mentioned in Section 1.2 that the translation method should be used experimentally by the combinatorialist to discover a direct description of the bijection produced. To make that easier, we introduce the following function for displaying bijections:

```
show≡ : ∀ {A B : Expr} → A ≡ B → IO ⊤
```

It simply lifts the given identity $A \equiv B$ to a bijection, then enumerates all the elements of the lifted A set using the `Expr`'s `generate` function that we introduced before, and then prints the current element and the corresponding element in the lifted B set according to the lifted bijection.[3]

While the above function may help one guess a direct bijection between the lifted A and B sets, it will still just be a conjecture. This would then need to be proved separately. However, as a first step, it may be useful to search for counterexamples. To assist with that, we introduce the following function:

```
check≡ : ∀ {A B : Expr} → A ≡ B → (lift A → lift B) → IO ⊤
```

It takes in an identity $A \equiv B$, as well as the conjectured bijection from the lifted A set to the lifted B set, then lifts the identity to a bijection, and exhaustively checks all elements of the lifted A set, again using the `generate` function introduced before, to look for elements whose image under the lifted bijection does not match the image under the conjectured bijection. Every counterexample found is then printed to the screen.

Usually it is not feasible to check all elements using the `check≡` function. Instead, one will usually run this for a short time, and if no counterexamples are found, then go on to attempt to prove the bijection. There are a few ways to proceed:

- Prove that the function is a bijection using existing combinatorial techniques.
- Prove that the function is a bijection using Agda:
  - Implement the function under the $\cong$ type. This will prove that the function is a bijection, but not necessarily the same bijection as the one produced by the translation method. Since the end goal is to find a direct bijection between the two given sets, it usually doesn't matter if the conjectured bijection is the same as the one given by the translation method, but perhaps that is not true in all cases.
  - If `f` is the bijection produced by the translation method and `g` is the conjectured bijection, prove that `f P≡ g`. Unfortunately, it is impossible to prove such an assertion in Agda, as Martin-Löf type theory—which Agda is based on—lacks what is known as functional extensionality [19, 35]. Luckily it is possible to prove a weaker condition, which is sufficient for our purposes, namely:

    ```
    ∀ x → f x P≡ g x
    ```

    This shows that the functions have the same outputs for all possible inputs, meaning that the functions are the same, and that we have the correct bijection.

[3] The return type, `IO ⊤`, represents a sequence of input/output actions, which in this case correspond to printing out the bijection.

One final thing that is suboptimal in our current implementation is when one needs to give an exhaustive definition of a bijection. This happens frequently when defining base cases for use with induction. In those cases, the $A \cong B$ type has to be instantiated by listing all the elements from the lifted A set and their corresponding projected elements from the lifted B set, and then once more the other way around, and then twice more to prove that the previous two mappings do indeed form a bijection. Consider the first base case given in Example 9. This would currently be implemented as follows in Agda:

```
thrice0 : nat 3 * fib 2 ≡ fib 4 + fib 0
thrice0 = proof Prefl (mkBij to from to-from from-to)
  where
    to : lift (nat 3 * fib 2) → lift (fib 4 + fib 0)
    to (nothing                , [] ::1 ::1) = inj₁ ([] ::1 ::1 ::1 ::1)
    to (nothing                , [] ::2)     = inj₁ ([] ::1 ::1 ::2)
    to (just nothing           , [] ::1 ::1) = inj₁ ([] ::2 ::1 ::1)
    to (just nothing           , [] ::2)     = inj₁ ([] ::2 ::2)
    to (just (just nothing) , [] ::1 ::1) = inj₁ ([] ::1 ::2 ::1)
    to (just (just nothing) , [] ::2)     = inj₂ []
    to (just (just (just ())) , _)

    from : lift (fib 4 + fib 0) → lift (nat 3 * fib 2)
    from (inj₁ ([] ::1 ::1 ::1 ::1)) = nothing                , [] ::1 ::1
    from (inj₁ ([] ::1 ::1 ::2))     = nothing                , [] ::2
    from (inj₁ ([] ::2 ::1 ::1))     = just nothing           , [] ::1 ::1
    from (inj₁ ([] ::2 ::2))         = just nothing           , [] ::2
    from (inj₁ ([] ::1 ::2 ::1))     = just (just nothing) , [] ::1 ::1
    from (inj₂ [])                   = just (just nothing) , [] ::2

    to-from : ∀ y → to (from y) P≡ y
    to-from (inj₁ ([] ::1 ::1 ::1 ::1)) = Prefl
    to-from (inj₁ ([] ::1 ::1 ::2))     = Prefl
    to-from (inj₁ ([] ::2 ::1 ::1))     = Prefl
    to-from (inj₁ ([] ::2 ::2))         = Prefl
    to-from (inj₁ ([] ::1 ::2 ::1))     = Prefl
    to-from (inj₂ [])                   = Prefl

    from-to : ∀ x → from (to x) P≡ x
    from-to (nothing                , [] ::1 ::1) = Prefl
    from-to (nothing                , [] ::2)     = Prefl
    from-to (just nothing           , [] ::1 ::1) = Prefl
    from-to (just nothing           , [] ::2)     = Prefl
    from-to (just (just nothing) , [] ::1 ::1) = Prefl
    from-to (just (just nothing) , [] ::2)     = Prefl
    from-to (just (just (just ())) , _)
```

And this is just the first base case—the second one is even bigger!

To make it easier to give an exhaustive definition of a bijection, we introduce the function `toBij`, which has the following type:

```
toBij : ∀ {A B : Expr} → List (lift A × lift B) → Maybe (lift A ≅ lift B)
```

That is, given two expressions A and B, and a list of pairs from the lifted A and B sets that represent the paired elements of the bijection, return a bijection between the lifted A and B sets if possible, or `nothing` otherwise.

To achieve this, we check that all elements from the lifted A and B sets are present in the list. This can be achieved using the `generate` and `exhaustive` functions we previously defined. Then we check that each element from the lifted A set is mapped to a distinct element from the lifted B set, and vice versa. If any of these checks fail, we return `nothing`, otherwise we have enough information to construct the bijection and prove that it is indeed a bijection.

The standard library in Agda has a function called `from-just`. It takes a value of type `Maybe A`, and if it is `just a`, then it returns `a`; otherwise it returns the (unique) value of type[4] `⊤`. This can be used in conjunction with the `toBij` function as follows:

```
thrice0 : nat 3 * fib 2 ≡ fib 4 + fib 0
thrice0 = proof Prefl (from-just (toBij {nat 3 * fib 2}
                                        {fib 4 + fib 0} (
    ((nothing             , [] ::1 ::1) , inj₁ ([] ::1 ::1 ::1 ::1)) L::
    ((nothing             , [] ::2)     , inj₁ ([] ::1 ::1 ::2)) L::
    ((just nothing        , [] ::1 ::1) , inj₁ ([] ::2 ::1 ::1)) L::
    ((just nothing        , [] ::2)     , inj₁ ([] ::2 ::2)) L::
    ((just (just nothing) , [] ::1 ::1) , inj₁ ([] ::1 ::2 ::1)) L::
    ((just (just nothing) , [] ::2)     , inj₂ []) L:: L[]
  )))
```

First, notice how much simpler this definition is than the previous one. Second, even though it looks less formal than the previous definition, it is still completely sound. If the given list of pairs does not define a proper bijection (e.g. if one of the lines was missing, or a specific value was used more than once), then `toBij` would not be able prove that the input forms a bijection, in which case it would return `nothing`, which in turn leads to `from-just` returning `⊤`. However, since the `proof` constructor was expecting the second argument to be a bijection, the code would not even type check! Contrast this to most other programming languages, and the previous Maple implementation in particular, where this would most certainly compile, and then either fail at runtime or, even worse, go completely unnoticed, producing faulty results.

Now that we have all the necessary tools, we can proceed to give some real examples of using the translation method in Agda.

2.4 Using the translate module

Let's go through the two examples presented in Section 1.2, and show how they can be implemented in Agda using our translate module.

Example 19. Recall Example 8, where we applied the translation method to the identity $2^n \cdot 2^n = 4^n$. Looking back at the proof of this identity, notice that all

[4]The data type `⊤`, often known as unit, has only a single inhabitant: `tt`.

the proof steps are simple combinations of the identities presented in Table 1.2—and thus already implemented in our translate module—except for the step that uses the identity $2 \cdot 2 = 4$. So before we are able to prove the main identity in Agda, we will have to prove this smaller identity first.

Proving this identity using our definition of equivalence requires that we both prove the underlying identity on natural numbers, as well as provide a bijection between the lifted sets. Proving the underlying identity is trivial, and a simple `Prefl` will do. For the bijection, we will use the one presented in Example 8. Specifying this bijection becomes straightforward using our `toBij` function:

```
two-four : nat 2 * nat 2 ≡ nat 4
two-four = proof Prefl (from-just (toBij {nat 2 * nat 2} {nat 4} (
    ((nothing      , nothing)      , nothing) ∷
    ((nothing      , just nothing) , just nothing) ∷
    ((just nothing , nothing)      , just (just nothing)) ∷
    ((just nothing , just nothing) , just (just (just nothing))) ∷ []
  )))
```

And now on to the main identity. The function that we want to implement has the following type:

```
phi : ∀ {n} → 2^ n * 2^ n ≡ 4^ n
```

Just as we did in Example 8, we will proceed with induction (or recursion, if you will). Our base case, $n = 0$, is trivial to prove, and there is only one possible bijection. In Agda code,

```
phi {ℕzero} = proof Prefl (from-just (toBij {2^ 0 * 2^ 0} {4^ 0} (
    (([] , []) , []) ∷ []
  )))
```

Then, for the inductive case, we will proceed exactly as in Example 8, making use of the equational reasoning syntax, as well as the translate module equivalents of the identities from Table 1.2:

```
phi {ℕsuc n} = begin
    2^ (ℕsuc n) * 2^ (ℕsuc n)       ≡⟨ *-cong 2^-def 2^-def ⟩
    (nat 2 * 2^ n) * (nat 2 * 2^ n) ≡⟨ sym *-assoc ⟩
    ((nat 2 * 2^ n) * nat 2) * 2^ n ≡⟨ *-cong *-comm refl ⟩
    (nat 2 * (nat 2 * 2^ n)) * 2^ n ≡⟨ *-cong (sym *-assoc) refl ⟩
    ((nat 2 * nat 2) * 2^ n) * 2^ n ≡⟨ *-assoc ⟩
    (nat 2 * nat 2) * (2^ n * 2^ n) ≡⟨ *-cong two-four refl ⟩
    (nat 4) * (2^ n * 2^ n)         ≡⟨ *-cong refl (phi {n}) ⟩
    (nat 4) * (4^ n)                ≡⟨ sym 4^-def ⟩
    4^ (ℕsuc n)                     ∎
```

This is all we need to prove our identity using the translate module. We can now use this identity in two ways; get a proof of the underlying identity on natural numbers using the `equality` function:

```
ℕphi : ∀ {n} → ℕ2^ n ℕ* ℕ2^ n P≡ ℕ4^ n
ℕphi {n} = equality (phi {n})
```

or, more importantly, apply the translation method to our proof of the identity to get a bijection between the sets $\mathbf{2}^n \times \mathbf{2}^n$ and $\mathbf{4}^n$, by using the `bijection` function:

```
Bphi : ∀ {n} → (BinStr n × BinStr n) ≅ QuadStr n
Bphi {n} = bijection (phi {n})
```

As an example, if we run[5]

```
getTo Bphi (Fzero ∷ Fzero ∷ [] , Fzero ∷ Fsuc Fzero ∷ [])
```

we get `Fzero ∷ Fsuc Fsuc Fzero ∷ []`, which agrees with our previous results from Table 1.3. Indeed, what we have here is the exact same bijection as we got in Example 8.

Example 20. Let us now continue with Example 9, where we applied the translation method to the identity $3F_n = F_{n+2} + F_{n-2}$. As we have mentioned before, it's a bit cumbersome to work with negative indices and subtraction when using the translate module, so we will instead prove a shifted version of this identity: $3F_{n+2} = F_{n+4} + F_n$. The type of the function that we want to implement is then as follows:

```
thrice : ∀ {n} → nat 3 * fib (ℕsuc (ℕsuc n))
                ≡ fib (ℕsuc (ℕsuc (ℕsuc (ℕsuc n)))) + fib n
```

Again we will closely follow our original application of the translation method, starting with the base cases:

```
thrice {0} = proof Prefl (from-just (toBij
  {nat 3 * fib (ℕsuc (ℕsuc 0))}
  {fib (ℕsuc (ℕsuc (ℕsuc (ℕsuc 0)))) + fib 0}
  (
    ((nothing              , [] ∷1 ∷1) , inj₁ ([] ∷1 ∷1 ∷1 ∷1)) ∷
    ((nothing              , [] ∷2)    , inj₁ ([] ∷1 ∷1 ∷2)) ∷
    ((just nothing         , [] ∷1 ∷1) , inj₁ ([] ∷2 ∷1 ∷1)) ∷
    ((just nothing         , [] ∷2)    , inj₁ ([] ∷2 ∷2)) ∷
    ((just (just nothing) , [] ∷1 ∷1) , inj₁ ([] ∷1 ∷2 ∷1)) ∷
    ((just (just nothing) , [] ∷2)    , inj₂ []) ∷ []
  )))

thrice {1} = proof Prefl (from-just (toBij
  {nat 3 * fib (ℕsuc (ℕsuc 1))}
  {fib (ℕsuc (ℕsuc (ℕsuc (ℕsuc 1)))) + fib 1}
  (
    ((nothing              , [] ∷1 ∷1 ∷1) , inj₁ ([] ∷1 ∷1 ∷1 ∷1 ∷1)) ∷
    ((nothing              , [] ∷1 ∷2)    , inj₁ ([] ∷1 ∷1 ∷1 ∷2)) ∷
```

[5]The `getTo` function returns the underlying function of the bijection data type. Similarly `getFrom` returns its inverse.

```
    ((nothing             , [] ∷2 ∷1)    , inj₁ ([] ∷1 ∷1 ∷2 ∷1)) ∷
    ((just nothing        , [] ∷1 ∷1 ∷1) , inj₁ ([] ∷2 ∷1 ∷1 ∷1)) ∷
    ((just nothing        , [] ∷1 ∷2)    , inj₁ ([] ∷2 ∷1 ∷2)) ∷
    ((just nothing        , [] ∷2 ∷1)    , inj₁ ([] ∷2 ∷2 ∷1)) ∷
    ((just (just nothing) , [] ∷1 ∷1 ∷1) , inj₁ ([] ∷1 ∷2 ∷1 ∷1)) ∷
    ((just (just nothing) , [] ∷1 ∷2)    , inj₁ ([] ∷1 ∷2 ∷2)) ∷
    ((just (just nothing) , [] ∷2 ∷1)    , inj₂ ([] ∷1)) ∷ []
  )))
```

The inductive case is then also a direct translation of our original proof into Agda code:

```
thrice {ℕsuc (ℕsuc n)} =
  begin
    nat 3 * fib (4 ℕ+ n)
  ≡⟨ *-cong refl fib-def ⟩
    nat 3 * (fib (3 ℕ+ n) + fib (2 ℕ+ n))
  ≡⟨ distribˡ-*-+ ⟩
    nat 3 * fib (3 ℕ+ n) + nat 3 * fib (2 ℕ+ n)
  ≡⟨ +-cong thrice thrice ⟩
    (fib (5 ℕ+ n) + fib (1 ℕ+ n)) + (fib (4 ℕ+ n) + fib n)
  ≡⟨ +-assoc ⟩
    fib (5 ℕ+ n) + (fib (1 ℕ+ n) + (fib (4 ℕ+ n) + fib n))
  ≡⟨ +-cong refl +-comm ⟩
    fib (5 ℕ+ n) + ((fib (4 ℕ+ n) + fib n) + fib (1 ℕ+ n))
  ≡⟨ +-cong refl +-assoc ⟩
    fib (5 ℕ+ n) + (fib (4 ℕ+ n) + (fib n + fib (1 ℕ+ n)))
  ≡⟨ +-cong refl (+-cong refl +-comm) ⟩
    fib (5 ℕ+ n) + (fib (4 ℕ+ n) + (fib (1 ℕ+ n) + fib n))
  ≡⟨ sym +-assoc ⟩
    (fib (5 ℕ+ n) + fib (4 ℕ+ n)) + (fib (1 ℕ+ n) + fib n)
  ≡⟨ +-cong (sym fib-def) (sym fib-def) ⟩
    fib (6 ℕ+ n) + fib (2 ℕ+ n)
  ∎
```

As in the previous example, we can now get access to the bijection produced by the translation method simply by calling the `bijection` function.

2.5 A semiring solver

Earlier we proved that the set `Expr`, equipped with the operations `+` and `*`, forms a commutative semiring under our definition of equivalence. It is then natural to ask what properties of commutative semirings we can make use of, or—perhaps more relevant to us—what existing tools for commutative semirings can we make use of?

There were a couple of tools that we found particularly interesting:

- A generic solver for fixpoint equations over semirings called FPsolve [10]. Unfortunately we have not yet found interesting use cases for fixed points in our semiring.

- The possibility of solving systems of linear equations in so-called closed semirings [9, 18]. Our semiring is not closed, as it is missing the required closure operator. One could try to find such an operator, but we have not pursued this further.

- Tools for automatically proving equalities in commutative rings or semirings, often known as ring or semiring solvers.

Since most of the work when using our translate module goes into proving identities, the last tool was by far the most interesting. In hope of making this part of using the translation method easier, we decided to bring semiring-solving capabilities to our translate module.

From a bird's-eye view, a semiring solver works as follows: Given two expressions A and B from the semiring, the goal is to provide a proof that $A \approx B$, where $\approx$ is the associated equivalence relation, or fail when that is not possible. To do that, the semiring solver brings both of the expressions to a common normal form. If their normal forms are the same, then proving that the two expressions are equivalent is trivial, but otherwise the solver fails.

It turns out that, when the two expressions only involve the elementary operations, constants and variables, the two expressions will always have the same normal form, and thus the semiring solver will succeed. As we will later see, things get a bit trickier when more complex constructs, such as recursion, are used, which may cause the normal forms of equivalent expressions to be distinct, leading to the solver failing. For now we will ignore these trickier cases.

The latest and greatest in semiring solvers seems to be the one described and implemented by Grégoire and Mahboubi [15]. It is currently used in the Coq proof assistant[6], and is known as the `ring` tactic.

Their implementation strives to be as efficient as possible. They achieve that by writing efficient algorithms, but also by using efficient data structures. In particular, to efficiently represent polynomials of high degree in their normal form, they use something known as a sparse Horner normal form, which they also describe in [15].

This kind of efficiency may be overkill for our purposes, as—at least so far—our identities have been rather simple. Either way, as we are using Agda and their implementation is written in Coq, it would be quite an effort for us to use their implementation. Luckily, some work towards porting their implementation to Agda has already been done [38]. The Agda implementation works, but does not make use of the sparse Horner normal form, and so is not as efficient as the original implementation.

Agda's `RingSolver` makes it easy to instantiate the solver using different semirings. Unfortunately, after trying to do that, we discovered two issues that would prohibit us from using their implementation, which both were related to our (necessary) representation of an expression:

[6]Coq is an interactive theorem prover, similar in many respects to Agda. [17]

- The solver requires support for decidable equality between elements of the semiring. As the elements in our semiring are themselves expressions, deciding this becomes almost as hard as implementing a ring solver in the first place.

- The solver does not normalize constants. If one considers a semiring such as the natural numbers coupled with addition and multiplication, if two constants are the same, they will both reduce to the exact same representation in Agda: both $0 + 2$ and $1 + 1$ reduce to 2, or `suc (suc zero)`. This, however, is not the case for the elements in our semiring: while $0 + 2$ and $1 + 1$ are equivalent under our equivalence relation, neither of them reduces any further. This prevents the solver from proving identities as simple as $0 + 2 = 1 + 1$ in our semiring.

Both of these issues could potentially be fixed by doing some major changes to their implementation. Instead of doing that, we decided to write our own implementation, specifically for our semiring. As the implementation would be specialized to our semiring it would certainly be a simpler implementation than their general one. And furthermore, this would allow us to add extensions to the solver specifically for our purposes, as we will later see.

Our implementation still follows the general approach of their implementation, and also makes use of the same interface. To describe this interface, and to give a better idea about how the solver will work, let us have a look at how one uses Agda's `RingSolver`.

Example 21. To prove the identity $(a+b)^2 = a^2+2ab+b^2$ in Agda, where a and b are natural numbers, one would need to perform a series of steps involving distributivity, and associativity and commutativity of both addition and multiplication. The semiring solver, however, can prove this for us almost completely automatically.

To use the solver, we first need to import the required modules, in this case the `SemiringSolver` for natural numbers:

```
open Data.Nat.Properties.SemiringSolver
  using (solve; _:=_; con; _:+_; _:*_)
```

The solver can now be used to prove the above identity as follows:

```
sq : ∀ a b → (a + b) * (a + b) ≡ a * a + 2 * a * b + b * b
sq = solve 2 (λ x y → (x :+ y) :* (x :+ y)
                   := x :* x :+ con 2 :* x :* y :+ y :* y)
           refl
```

Let us dissect this code a bit. First we have a type signature that represents the proposition we want to prove. Then, in the body of the function where a proof of the proposition would usually follow, we just a have single call to the `solve` function. This function takes three arguments:

- The first argument is a natural number n representing the number of variables in the equality we want to prove. In our case the variables are a and b, so we pass the argument 2.

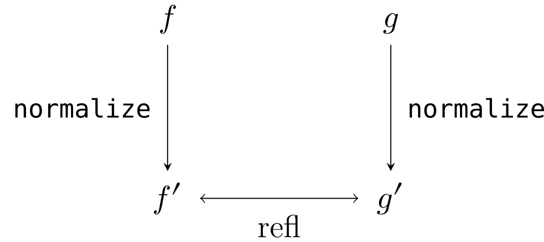

Figure 2.1: A depiction of the semiring solver. Given expressions f and g, they are normalized, and then the user is expected to provide a proof that the normalized versions are equivalent.

- The second argument is a function which takes in n arguments, each representing a separate variable, and then returns an abstract syntax tree representing the equality to be proved. The syntax tree has constructors corresponding to both addition and multiplication, both prefixed with a colon, as well as a constructor `con` specifically for constants (i.e. expressions not containing any of the variables). Finally, `:=` is used to separate the expression on the left from the expression on the right.

- The final argument, `refl` in this case, requires a bit of explaining. Its type is not known until both the first and second arguments to `solve` have been passed. At that point, the semiring solver goes off and does most of the work that it needs to do to prove the given equality. Specifically, if f and g are the expressions on the left and right sides of the equality, respectively, then it goes ahead and normalizes each of the expressions, producing new expressions f' and g' in normal form for f and g, respectively. It does this very carefully, internally resulting in a proof that $f = f'$ and that $g = g'$. The final step is then to prove that $f' = g'$, and this step it leaves up to the caller of `solve`. And this is exactly what it expects as the third argument: a proof that $f' = g'$. The whole process is depicted in Figure 2.1.

  In this case, and in all cases where the expressions are sufficiently simple, the normalized expressions are identical, meaning that reflexivity is enough to prove that they are equivalent, which is why we pass in `refl` as the final argument.

The `solve` function finally returns the required proof.

We will be using the same interface for our solver, so solving an identity such as $(a + b)^2 = a^2 + 2ab + b^2$ in our semiring would look almost identical. In fact, Agda's implementation was very modular, allowing us to easily hook into and reuse their interface.

What remains, then, is to implement the normalization. This consists of two parts; actually performing the normalization, and then proving that the original and the normalized expressions are equivalent. To perform the normalization we will need to know how the expressions are structured. In Agda's implementation, this is why they require the second argument: the expression represented as an abstract syntax tree. We already have that to a great extent with the `Expr` type, but there are still a couple of things missing:

- Arguments to $F_\circ$ and 2° are still represented as natural numbers, which means we don't have any information about how they are structured.

- We need to be able to represent variables as part of our abstract syntax trees.

We resolved the above two issues by introducing the following augmented `Expr` type[7]:

```
mutual
  data :Fun (m : ℕ) : Set where
    :2^' : (n : :Expr m) → :Fun m
    :fib' : (n : :Expr m) → :Fun m

  data :Expr (n : ℕ) : Set where
    :zero : :Expr n
    :suc : (x : :Expr n) → :Expr n

    _:+_ : (l r : :Expr n) → :Expr n
    _:*_ : (l r : :Expr n) → :Expr n

    :fun : (f : :Fun n) → :Expr n

    :var : Fin n → :Expr n
```

Here we follow the convention used in Agda's solver where types and constructors belonging to the abstract syntax tree are prefixed with a colon. There are a few other things to make note of:

- The `:Expr` type is parameterized by a natural number n, the number of variables in the expression. Furthermore it is now also possible to represent variables using the `:var` constructor. We will number the variables consecutively from 0 to $n-1$, and `:var` i will be used to represent the i-th variable.

- Functions such as F_n and 2^n are constructors of a separate type called `:Fun`. This is to make it simpler to add support for new functions. Unfortunately it also means that one would have to prefix all references to these functions with a `:fun`, but that can be easily circumvented with the following wrapper functions:

  ```
  :2^ : ∀ {m} → :Expr m → :Expr m
  :2^ n = :fun (:2^' n)

  :fib : ∀ {m} → :Expr m → :Expr m
  :fib n = :fun (:fib' n)
  ```

- The same functions now take in expressions as arguments instead of natural numbers.

To reason about these abstract syntax trees in terms of their corresponding expressions, we also need to define the semantics of these syntax trees, i.e. describe how an abstract syntax tree is to be interpreted as an expression. When variables are involved, we

[7]Definitions that mutually depend on each other need to be specified within a `mutual` block.

also need to have the values of the variables to be able to substitute them into the expression. We call this the environment, often denoted with Γ, and will represent it as a vector of expressions of length n, where n is the number of variables, as follows:

```
Env : ℕ → Set
Env n = Vec Expr n
```

The expression at position i in the vector will then represent the value of variable number i. The semantics can then be defined as follows in terms of the regular `Expr` type:

```
mutual
  ⟦_⟧F : ∀ {n} → (x : :Fun n) → (Γ : Env n) → Expr
  ⟦ :fib' n ⟧F Γ = fib (value (⟦ n ⟧ Γ))
  ⟦ :2^' n ⟧F Γ = 2^ (value (⟦ n ⟧ Γ))

  ⟦_⟧ : ∀ {n} → (x : :Expr n) → (Γ : Env n) → Expr
  ⟦ :var x ⟧ Γ = lookup x Γ
  ⟦ :zero ⟧ Γ = zero
  ⟦ :suc x ⟧ Γ = suc (⟦ x ⟧ Γ)
  ⟦ l :+ r ⟧ Γ = ⟦ l ⟧ Γ + ⟦ r ⟧ Γ
  ⟦ l :* r ⟧ Γ = ⟦ l ⟧ Γ * ⟦ r ⟧ Γ
  ⟦ :fun f ⟧ Γ = (⟦ f ⟧F Γ)
```

With these types to represent abstract syntax trees, and their semantics defined, we are now ready to proceed to the actual normalization.

So what does the normal form look like? First, as we noted above, for now we will ignore tricky cases emanating from things like recursion. To do that, we will treat functions as black boxes. In a sense they will be treated as variables, where two functions are represented with the same variable if and only if they are the same function and have the same arguments.

So how do we know if two functions have the same arguments? We will answer this recursively, by bringing all function arguments to normal form as well. We will have a closer look at this part later, but for now we can assume that the function arguments are already normalized.

What now remains are only variables and constants, as well as addition and multiplication. The intuition is that this will always be a polynomial in multiple variables. However, the current representation has a lot of ambiguity, and there will be many possible ways to represent the same polynomial. Let's start developing the normalization process, and along the way we will converge to a normal form for polynomials.

Now, to make it as easy as possible to implement, each step in our normalization process will remove one source of ambiguity from the expression, and is roughly described as follows:

- Define a new data type for representing an abstract syntax tree that is in some sense more normalized than the one in the previous step.
- Define the semantics of this representation in terms of `Expr`.

- Create a function that proceeds one step in the normalization process, i.e. takes in an expression represented as the data type in the previous step, and returns an expression represented as the data type introduced in this step.
- Prove that this function is sound, i.e. that it preserves the semantics of the input expression.

So where does this ambiguity come from? Consider the expressions $a + b$ and $b + a$. They are syntactically distinct, yet equivalent by the commutativity of addition. Similarly each of the properties that a commutative semiring satisfies seems to introduce ambiguity. Let's go over each of the properties and see how we will get rid of the ambiguity introduced by the respective property:

- The additive and multiplicative identities: $a + 0 = a$ and $a \cdot 1 = a$. To remove the ambiguity they introduce, when we see $a + 0$ or $a \cdot 1$ in the syntax tree, we will replace that particular subexpression with a.
- Annihilation when multiplying by zero: $a \cdot 0 = 0$. To remove the ambiguity this introduces, when we see $a \cdot 0$ in the syntax tree, we will replace that particular subexpression with 0.
- Commutativity of addition and multiplication: $a + b = b + a$ and $a \cdot b = b \cdot a$. To remove the ambiguity they introduce, we will introduce a total order $\preccurlyeq$ on expressions, and force $a \preccurlyeq b$ to be satisfied every time we have either $a + b$ or $a \cdot b$ in the syntax tree, swapping the expressions when necessary.
- Associativity of addition and multiplication: $(a + b) + c = a + (b + c)$ and $(a \cdot b) \cdot c = a \cdot (b \cdot c)$. To remove the ambiguity they introduce, when we see either $(a + b) + c$ or $(a \cdot b) \cdot c$ in the syntax tree, we will replace that particular subexpression with $a + (b + c)$ and $a \cdot (b \cdot c)$, respectively.
- Distributivity of multiplication over addition: $a \cdot (b + c) = a \cdot b + a \cdot c$. To remove the ambiguity this introduces, when we see $a \cdot (b + c)$ in the syntax tree, we will replace that particular subexpression with $a \cdot b + a \cdot c$.

We will resolve these ambiguities one at a time, starting with distributivity.

In this first step we will take as input an arbitrary syntax tree, `:Expr n`, and convert it to a representation where distributivity no longer has any effect. To find such a representation, consider what happens when we repeatedly apply the rule suggested above of replacing each occurrence of $a \cdot (b + c)$ with $a \cdot b + a \cdot c$, expanding the expression as much as possible. What we end up with is a sum of monomials, where each monomial is a product of constants, variables and functions, e.g.

$$(5 \cdot (a + b) + c) \cdot 2^n \text{ becomes } ((5 \cdot a) \cdot 2^n + (5 \cdot b) \cdot 2^n) + c \cdot 2^n$$

The natural representation is thus a sum of monomials, which we implemented as follows:

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

```
  _:+_ : Constant → Constant → Constant
  _:*_ : Constant → Constant → Constant

data Monomial (n : ℕ) : Set₂ where
  con : Constant → Monomial n
  var : Fin n → Monomial n
  fun : :Fun n → Monomial n
  _:*_ : Monomial n → Monomial n → Monomial n

data SumOfMonomials (n : ℕ) : Set₂ where
  mon : Monomial n → SumOfMonomials n
  _:+_ : SumOfMonomials n → SumOfMonomials n → SumOfMonomials n
```

We further define the following semantics for the representation:

```
⟦_⟧C : Constant → Expr
⟦ :zero ⟧C = zero
⟦ :suc x ⟧C = suc ⟦ x ⟧C
⟦ x :+ y ⟧C = ⟦ x ⟧C + ⟦ y ⟧C
⟦ x :* y ⟧C = ⟦ x ⟧C * ⟦ y ⟧C

⟦_⟧M : ∀ {n} → Monomial n → Env n → Expr
⟦ con x ⟧M Γ = ⟦ x ⟧C
⟦ var x ⟧M Γ = ⟦ :var x ⟧ Γ
⟦ fun f ⟧M Γ = ⟦ :fun f ⟧ Γ
⟦ x :* y ⟧M Γ = ⟦ x ⟧M Γ * ⟦ y ⟧M Γ

⟦_⟧SM : ∀ {n} → SumOfMonomials n → Env n → Expr
⟦ mon x ⟧SM Γ = ⟦ x ⟧M Γ
⟦ x :+ y ⟧SM Γ = ⟦ x ⟧SM Γ + ⟦ y ⟧SM Γ
```

Now we just need a function that carries out the expansion itself, i.e. a function of the following type:

```
distrib : ∀ {n} → :Expr n → SumOfMonomials n
```

It can be implemented as follows:

```
-- A helper function: given p and q, expands p * q
*-distrib : ∀ {n} → (p q : SumOfMonomials n) → SumOfMonomials n
*-distrib (mon l) (mon r) = mon (l :* r)
*-distrib (mon l) (r₁ :+ r₂) = *-distrib (mon l) r₁ :+ *-distrib (mon l) r₂
*-distrib (l₁ :+ l₂) (mon r) = *-distrib l₁ (mon r) :+ *-distrib l₂ (mon r)
*-distrib (l₁ :+ l₂) (r₁ :+ r₂)
    = *-distrib l₁ r₁ :+ *-distrib l₁ r₂ :+ *-distrib l₂ r₁ :+ *-distrib l₂ r₂

distrib : ∀ {n} → :Expr n → SumOfMonomials n
distrib (:var x) = mon (var x)
distrib :zero = mon (con :zero)
```

```
distrib (:suc x) = mon (con (:suc :zero)) :+ distrib x
distrib (l :+ r) = distrib l :+ distrib r
distrib (l :* r) = *-distrib (distrib l) (distrib r)
distrib (:fun f) = mon (fun f)
```

We also need to prove that the function is sound. That is, we want all expressions to have the same semantics before and after the function is called, no matter what the values of the variables are (i.e. irrespective of the environment):

```
distrib-correct : ∀ {n}
                → (Γ : Env n)
                → (p : :Expr n)
                → ⟦ distrib p ⟧SM Γ ≡ ⟦ p ⟧ Γ
```

We omit the proof as it is a bit long, but mostly mechanical.

Let's now deal with associativity of addition, using the rule proposed above. After we have applied this rule, our sum of monomials will now be totally associated to the right, i.e. the subtree induced by vertices in our expression tree containing a plus will be right-leaning:

$$((5 \cdot a) \cdot 2^n + (5 \cdot b) \cdot 2^n) + c \cdot 2^n \text{ becomes } (5 \cdot a) \cdot 2^n + ((5 \cdot b) \cdot 2^n + c \cdot 2^n)$$

This can be represented as follows:

```
data RightLeaningSumOfMonomials (n : ℕ) : Set₂ where
  nil : RightLeaningSumOfMonomials n
  _:+_ : Monomial n
       → RightLeaningSumOfMonomials n
       → RightLeaningSumOfMonomials n

⟦_⟧RLSM : ∀ {n} → RightLeaningSumOfMonomials n → Env n → Expr
⟦ nil ⟧RLSM Γ = zero
⟦ l :+ r ⟧RLSM Γ = ⟦ l ⟧M Γ + ⟦ r ⟧RLSM Γ
```

Then we just need a function to convert a sum of monomials to right-leaning sum of monomials:

```
lean-right : ∀ {n} → SumOfMonomials n → RightLeaningSumOfMonomials n
```

and a proof that the function is sound:

```
lean-right-correct : ∀ {n}
                   → (Γ : Env n)
                   → (p : SumOfMonomials n)
                   → ⟦ lean-right p ⟧RLSM Γ ≡ ⟦ p ⟧SM Γ
```

We will skip their implementations for brevity.

If we now focus on the individual monomials, there is still a lot of ambiguity going on there; associativity and commutativity of multiplication between the variables,

functions and constants, and the constants are still structured somewhat arbitrarily. Let's first deal with associativity between the three kinds of terms, and ignore the constants for now.

To represent a monomial where associativity is ignored we will introduce what we call a semi-normalized monomial:

```
data SnormalizedMonomial (n : ℕ) : Set₂ where
  mon : Constant
      → (vs : List (Fin n))
      → (fs : List (:Fun n))
      → SnormalizedMonomial n

⟦_⟧LV : ∀ {n} → (cs : List (Fin n)) → (Γ : Env n) → Expr
⟦ [] ⟧LV = suc zero
⟦ x ∷ xs ⟧LV = var x * ⟦ xs ⟧LV Γ

⟦_⟧LF : ∀ {n} → (cs : List (:Fun n)) → (Γ : Env n) → Expr
⟦ [] ⟧LF = suc zero
⟦ x ∷ xs ⟧LF = fun x * ⟦ xs ⟧LF

⟦_⟧SNM : ∀ {n} → SnormalizedMonomial n → Env n → Expr
⟦ mon c vs fs ⟧SNM Γ = ⟦ c ⟧C * (⟦ vs ⟧LV Γ * ⟦ fs ⟧LF Γ)
```

Here the variables and functions are stored as lists, meaning that associativity does not play a role, and then the constant is stored separately. To semi-normalize monomials from previous steps we introduce the following two functions:

```
snormalize-monomial : ∀ {n} → Monomial n → SnormalizedMonomial n

snormalize-monomial-correct : ∀ {n}
                            → (Γ : Env n)
                            → (p : Monomial n)
                            → ⟦ snormalize-monomial p ⟧SNM Γ ≡ ⟦ p ⟧M Γ
```

Let's continue working on monomials, and now consider commutativity of multiplication. Notice that any permutation of the variables and the functions will give a different `SnormalizedMonomial` representation, even though they are all equivalent. To fix that, we will sort both the list of variables and the list of functions, according to some total order on the respective types. This will bring us down to a unique representation for this class of expressions.

Obviously the choice of which total order to use is irrelevant, as long as distinct expressions are also considered distinct by the order. Thus, for variables, choosing such a total order is easy: as the variables are represented with natural numbers, we can use the standard order on natural numbers. Coming up with a total order for the function types is a bit trickier, as their arguments could be arbitrary expressions (although they are already in normal form, as we previously assumed). With that in mind, a simple total order is the lexicographic order of the expressions' textual representations. We used a variant of this which was a bit easier to implement.

Making a function that sorted each of the lists according to the respective total orders, and then proving that it was sound, was a non-trivial exercise. However, the choice of a sorting algorithm seemed to make a big difference, and using the insertion sort algorithm made it relatively straightforward.

With these things figured out we implemented two functions to sort the respective lists, and then a function that applied both of them:

```
sort-lv : ∀ {n} → List (Fin n) → List (Fin n)
sort-lv-correct : ∀ {n}
               → (Γ : Env n)
               → (xs : List (Fin n))
               → ⟦ sort-lv xs ⟧LV Γ ≡ ⟦ xs ⟧LV Γ

sort-lf : ∀ {n} → List (:Fun n) → List (:Fun n)
sort-lf-correct : ∀ {n}
                 → (Γ : Env n)
                 → (xs : List (:Fun n))
                 → ⟦ sort-lf xs ⟧LF Γ ≡ ⟦ xs ⟧LF Γ

sort-snormalized-monomial : ∀ {n} → SnormalizedMonomial n
                                  → SnormalizedMonomial n
sort-snormalized-monomial-correct
    : ∀ {n}
    → (Γ : Env n)
    → (p : SnormalizedMonomial n)
    → ⟦ sort-snormalized-monomial p ⟧SNM Γ ≡ ⟦ p ⟧SNM Γ
```

The final ambiguity within the monomials arises from the constants, as they can still be structured arbitrarily. We have already seen how to represent such constants uniquely: the unary representation of Peano arithmetic used for natural numbers in Agda! Defining such a representation is trivial:

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

⟦_⟧NC : ∀ {n} → NormalizedConstant → Env n → Expr
⟦ :zero ⟧NC Γ = zero
⟦ :suc x ⟧NC Γ = suc ⟦ x ⟧NC Γ
```

Then we define the following functions to perform the normalization of constants:

```
normalize-constant : Constant → NormalizedConstant

normalize-constant-correct : ∀ {n}
                            → (Γ : Env n)
                            → (p : Constant)
                            → ⟦ normalize-constant p ⟧NC Γ ≡ ⟦ p ⟧C
```

Now that we have the ability to normalize constants, we will update the representation of our monomials to include only normalized constants:

```
data NormalizedMonomial (n : ℕ) : Set₂ where
  mon : NormalizedConstant
      → (vs : List (Fin n))
      → (fs : List (:Fun n))
      → NormalizedMonomial n

⟦_⟧NM : ∀ {n} → NormalizedMonomial n → Env n → Expr
⟦ mon x vs fs ⟧NM = ⟦ x ⟧NC Γ * (⟦ vs ⟧LV Γ * ⟦ fs ⟧LF Γ)
```

And then, of course, the associated normalization functions:

```
normalize-constants : ∀ {n} → SnormalizedMonomial n → NormalizedMonomial n

normalize-constants-correct : ∀ {n}
                            → (Γ : Env n)
                            → (p : SnormalizedMonomial n)
                            → ⟦ normalize-constants p ⟧NM Γ ≡ ⟦ p ⟧SNM Γ
```

Similarly, we will now update our right-leaning sum of monomials to use the normalized versions of the monomials:

```
data RightLeaningSumOfNormalizedMonomials : (n : ℕ) → Set₂ where
  nil : ∀ {n} → RightLeaningSumOfNormalizedMonomials n
  _:+_ : ∀ {n} → NormalizedMonomial n
             → RightLeaningSumOfNormalizedMonomials n
             → RightLeaningSumOfNormalizedMonomials n

⟦_⟧RLSNM : ∀ {n} → RightLeaningSumOfNormalizedMonomials n → Env n → Expr
⟦ nil ⟧RLSNM Γ = zero
⟦ x :+ y ⟧RLSNM Γ = ⟦ x ⟧NM Γ + ⟦ y ⟧RLSNM Γ

normalize-monomials : ∀ {n} → RightLeaningSumOfMonomials n
                            → RightLeaningSumOfNormalizedMonomials n

normalize-monomials-correct : ∀ {n}
                            → (Γ : Env n)
                            → (p : RightLeaningSumOfMonomials n)
                            → ⟦ normalize-monomials p ⟧RLSNM Γ ≡ ⟦ p ⟧RLSM Γ
```

So now the monomials have been completely normalized: any two equivalent monomials will be represented exactly the same under our `NormalizedMonomial` type. What remains is to finish normalizing the expression as a whole.

As a step towards that, recall both the annihilating zero and the identity for addition: $a \cdot 0 = 0$ and $a + 0 = a$. Following the rules given above, we will remove any monomials which have a leading constant of 0. To do that, we first implemented the following

function to check if a monomial is zero (irrespective of the environment), and in that case give a proof of that fact:

```
is-zero? : ∀ {n} → (p : NormalizedMonomial n)
                → Maybe (∀ Γ → ⟦ p ⟧NM Γ ≡ zero)
```

With that function it was straightforward to create a function that removed monomials with a leading 0 constant:

```
throw-out-zeros : ∀ {n} → RightLeaningSumOfNormalizedMonomials n
                        → RightLeaningSumOfNormalizedMonomials n
```

Let's now look at the associativity of addition, which in our case means that monomials can be permuted arbitrarily without affecting the value of the expression. As with commutativity of multiplication, we will resolve this by sorting the monomials. We will use the same total order as we previously used, comparing the expressions lexicographically, and then use the insertion sort algorithm. This was implemented with the following function:

```
sort : ∀ {n} → RightLeaningSumOfNormalizedMonomials n
             → RightLeaningSumOfNormalizedMonomials n

sort-correct : ∀ {n} → (Γ : Env n)
                     → (p : RightLeaningSumOfNormalizedMonomials n)
                     → ⟦ sort p ⟧RLSNM Γ ≡ ⟦ p ⟧RLSNM Γ
```

The last ambiguity in our representation that we need to fix is a little subtle. Consider the expressions $2 \cdot x \cdot y + 3 \cdot x \cdot y$ and $5 \cdot x \cdot y$. Currently these are two distinct valid `RightLeaningSumOfNormalizedMonomials` representations of expressions, yet they are equivalent, because of distributivity. Contrary to the rule for distributivity that was presented above and used in the first step, we are going to fix this by combining monomials that only differ in their leading constants.

It is a bit cumbersome to walk through the monomials, looking for and gathering monomials that only have a different leading constant. However, we can alter the `sort` function that we just implemented to make this easier. That is, if we alter the total order used by the `sort` function to ignore the leading constant, then expressions that are the same except for their leading constant will be consecutive after sorting.

Now grouping together the required monomials becomes just a matter of iterating through the list, combining consecutive monomials whenever they only differ in their leading constant. This was implemented as the following function:

```
squash : ∀ {n} → RightLeaningSumOfNormalizedMonomials n
               → RightLeaningSumOfNormalizedMonomials n

squash-correct : ∀ {n} → (Γ : Env n)
                       → (xs : RightLeaningSumOfNormalizedMonomials n)
                       → ⟦ squash xs ⟧RLSNM Γ ≡ ⟦ xs ⟧RLSNM Γ
```

So now, finally, we have the required normal form, where all equivalent polynomials have the same representation. The last step is just to make use of the transformations, to make one function, `normalize`, that performs all of the normalization:

```
normalize : ∀ {n} → :Expr n → RightLeaningSumOfNormalizedMonomials n
normalize = squash ∘ sort ∘ throw-out-zeros ∘
                normalize-monomials ∘ lean-right ∘ distrib
```

Proving that this function preserves the semantics of the given expression is also just a simple matter of combining the soundness proofs of the individual steps, although the precise implementation is skipped for brevity:

```
normalize-correct : ∀ {n}
                    → (Γ : Env n)
                    → (x : :Expr n)
                    → ⟦ normalize x ⟧RLSNM Γ ≡ ⟦ x ⟧ Γ
```

With the `normalize` function implemented, and its soundness proved, we can plug those functions into Agda's semiring solver interface, and start using it:

Example 22. Consider the identity from Example 21: $(a+b)^2 = a^2 + 2ab + b^2$. This identity can now be proved in terms of our translate module. The implementation is almost the same as the previous example, with the only differences being related to the constant 2:

```
sq₂ : ∀ a b → (a + b) * (a + b) ≡ a * a + nat 2 * a * b + b * b
sq₂ = solve 2 (λ x y → (x :+ y) :* (x :+ y)
                     := x :* x :+ :nat 2 :* x :* y :+ y :* y)
            refl
```

This identity can then be lifted to a bijection between `(A ⊔ B) × (A ⊔ B)` and `(A × A) ⊔ (Fin 2 × A × B) ⊔ (B × B)` using the translation method, by simply calling the `bijection` function. As an example, Table 2.1 shows the bijection produced by calling `bijection (sq₂ 1 2)`.

This final example may raise an interesting question. Recall that, if the translation method is given a "natural" algebraic proof as input, the hope is that the bijection produced will also be natural. Thus one may wonder if using automatic methods to find an algebraic proof, as is presented here, leads to a natural bijection being produced. If one studies Table 2.1, this does indeed seem to be the case for that particular example. We will further study this question in Section 2.6.

2.5.1 Heuristics for the solver

As we have seen our solver can handle all polynomial expressions correctly, giving a proof if the two polynomials are equal, and failing otherwise. However, the more interesting identities usually involve more complex constructions such as functions and recursion. So far we have been treating these more complex expressions, namely functions, as black boxes. This of course severely limits the capabilities of the solver. Let's look at how we can make the solver more powerful, at least in certain cases.

lift ((a + b) * (a + b))	lift (a * a + 2 * a * b + b * b)
(inj₁ zero , inj₁ zero)	inj₁ (inj₁ (zero , zero))
(inj₁ zero , inj₂ zero)	inj₁ (inj₂ ((zero , zero) , zero))
(inj₁ zero , inj₂ one)	inj₁ (inj₂ ((zero , zero) , one))
(inj₂ zero , inj₁ zero)	inj₁ (inj₂ ((one , zero) , zero))
(inj₂ one , inj₁ zero)	inj₁ (inj₂ ((one , zero) , one))
(inj₂ zero , inj₂ zero)	inj₂ (zero , zero)
(inj₂ zero , inj₂ one)	inj₂ (zero , one)
(inj₂ one , inj₂ zero)	inj₂ (one , zero)
(inj₂ one , inj₂ one)	inj₂ (one , one)

Table 2.1: A part of the bijection produced by automatically proving the identity $(a+b)^2 = a^2 + 2ab + b^2$, with `nothing` denoted as `zero` and `just nothing` denoted as `one` for brevity.

Consider the identity $F_{n+3} = 2 \cdot F_{n+1} + F_n$. This identity is easily proved by applying the defining identity $F_n = F_{n-1} + F_{n-2}$ for Fibonacci numbers repeatedly on the left side of the equation. However, as far as the solver is concerned, proving this identity is equivalent to proving the identity $x = 2 \cdot y + z$, since functions with distinct parameters are essentially treated as distinct variables. This identity is obviously false, and the solver fails.

Now suppose that we ourselves apply the defining identity of Fibonacci numbers on the left side of the equation. Then we have the identity $(F_{n+1} + F_n) + F_{n+1} = 2 \cdot F_{n+1} + F_n$. This identity, on the other hand, is equivalent to the identity $(x+y) + x = 2 \cdot x + y$, as far as the solver is concerned. This identity is obviously true, and the solver will have no trouble coming up with a proof.

This example gives our heuristic for the solver: whenever it sees F_E, where E is any expression the solver can prove is equivalent to $n+2$, for some expression n, the solver will expand this to $F_{n+1} + F_n$. The same kind of heuristic can be applied to other functions, such as expanding 2^E to $2 \cdot 2^n$, whenever the solver can prove that E is equivalent to $n+1$.

So how does the solver determine if an expression is of the form $c + n$, for some constant c and some expression n? Recall that we assumed that function arguments have already been brought to normal form. Constants, of course, are monomials with no variables. Hence, as our normal form is a list of monomials, it is easy to scan through the list looking for the constant term.

To do that we introduce the `uncon` function:

```
uncon : ∀ {n} → :Expr n → NormalizedConstant × :Expr n
```

Given an expression, it brings the expression to normal form by calling the `normalize` function, and then returns the constant part as well as the remainder of the expression. As with the other functions related to our solver, it has an accompanying soundness proof:

```
uncon-correct : ∀ {n}
              → (Γ : Env n)
```

```
          → (x : :Expr n)
          → ⟦ proj₁ (uncon x) ⟧NC Γ + ⟦ proj₂ (uncon x) ⟧ Γ ≡ ⟦ x ⟧ Γ
```

With this function we can then implement our heuristic as the `expand` function:

```
expand : ∀ {n} → :Expr n → :Expr n
```

For most operations, such as addition and multiplication, it just applies the expansion recursively on the operands. For functions such as F_n, it uses `uncon` to apply our heuristic:

```
expand (:fun (:fib' n)) with uncon n
expand (:fun (:fib' n)) | :zero          , x = :fib ⟦ x ⇓⟧
expand (:fun (:fib' n)) | :suc :zero     , x = :fib ⟦ :suc :zero :+ x ⇓⟧
expand (:fun (:fib' n)) | :suc (:suc c) , x = ⟦ :fib (:suc (:⟦ c ⟧NC :+ x))
                                                  :+ :fib (:⟦ c ⟧NC :+ x) ⇓⟧
```

Here ⟦ `x` ⇓⟧ normalizes the expression `x`, and then gives the semantics of the result. Some details are skipped for brevity.

There are two things to note here. First, in the case that the constant is zero or one, the `expand` function applies normalization to the arguments of the resulting function. Second, in the case that the constant is at least two, it seems like the heuristic is only being applied once, but not repeatedly as should be the case if the constant is strictly larger than two, and also that normalization is being applied to the result of the heuristic.

This is where we need to—and indeed, can finally—tie together an open loop. That is, by making the `expand` function the first step in our normalization, two things happen. First, as the base cases of our heuristic recursively normalized the arguments of the resulting functions, all functions will now have normalized arguments. This resolves our assumption from the previous section about functions having arguments in normal form. And second, as the other case called normalization recursively on the result, any heuristics will be applied repeatedly.

Our `normalize` function will thus look as follows:

```
normalize : ∀ {n} → :Expr n → RightLeaningSumOfNormalizedMonomials n
normalize = squash ∘ sort ∘ throw-out-zeros ∘ normalize-monomials
                   ∘ lean-right ∘ distrib ∘ expand
```

The soundness of the new `normalize` function can then be updated with the soundness of the `expand` function:

```
expand-correct : ∀ {n} → (Γ : Env n)
                       → (e : :Expr n)
                       → ⟦ expand e ⟧ Γ ≡ ⟦ e ⟧ Γ
```

With the heuristic in place, the solver can now prove some more complex identities.

Example 23. Consider, again, the identity $3F_n = F_{n+2} + F_{n-2}$ from Examples 9 and 20. Similar to one of the examples given at the start of this section, the basic version of the solver will not be able to prove this identity. However, as it turns out, the solver is able to prove the identity after the heuristic has been added:

```
thrice' : ∀ {n} → nat 3 * fib (ℕsuc (ℕsuc n))
                ≡ fib (ℕsuc (ℕsuc (ℕsuc (ℕsuc n)))) + fib n
thrice' {n} rewrite Psym (nat-value n)
  = solve 1 (λ n → :nat 3 :* :fib (:suc (:suc n))
                := :fib (:suc (:suc (:suc (:suc n)))) :+ :fib n)
          refl
          (nat n)
```

Notice that calling the solver is as simple as copying the identity that we want to prove, and prefixing most of the terms with a colon to use their abstract syntax tree equivalents. One thing that needs some attention, however, is the argument to the `fib` function. In the original expression this argument is a natural number, while in the abstract syntax tree the semantics of this argument is an expression. Thus Agda will prove a slightly different result, and then complain that `value (nat n)` is not the same thing as `n`. But they actually are the same, as proved by the `nat-value` function which has the following type:

```
nat-value : ∀ n → value (nat n) P≡ n
```

Using the `rewrite` directive[8] we can convince Agda, using that proof, that they are indeed the same. After doing that, we have our proof, and can call the `bijection` function to apply the translation method and get a bijection. This bijection is given in Table 2.2.

Now contrast this with our original bijection, partially given in Table 1.4. If one determines the direct formulations for these two bijections, one will find that they are indeed different, although they do share some similarities and both look very natural. Also notice that the original proof uses induction, while the solver just applies the heuristic and does some basic algebraic manipulations. These results are quite interesting, and we will consider them further in the next section.

2.6 On the number of natural bijections

Recall Example 9 from Section 1.2, where we used the translation method to lift the identity $3F_n = F_{n+2} + F_{n-2}$ to a bijection. To do that, we used induction, along with a large and arbitrary-looking base case. The precise choice of a base case obviously affected the resulting bijection, and one could find that not all of the resulting bijections were, in some sense, as natural as others.

Wood and Zeilberger noticed that, by offsetting the recurrence, they could either make the base cases larger or smaller. To make the base cases as small as possible,

[8] To recap: Agda's `rewrite` directive takes an equality `a P≡ b` and replaces all occurrences of `a` in the return type with `b`. This is nothing more than syntactic sugar for a `with` statement with some pattern matching.

(x, s)	proj. of (x, s)
$(0, 1111)$	11112_{L}
$(0, 211)$	2112_{L}
$(0, 121)$	1212_{L}
$(0, 112)$	1122_{L}
$(0, 22)$	222_{L}
$(1, 1111)$	111111_{L}
$(1, 211)$	21111_{L}
$(1, 121)$	12111_{L}
$(1, 112)$	11211_{L}
$(1, 22)$	2211_{L}
$(2, 1111)$	11121_{L}
$(2, 211)$	2121_{L}
$(2, 121)$	1221_{L}
$(2, 112)$	11_{R}
$(2, 22)$	2_{R}
$(0, 11111)$	111112_{L}
$(0, 2111)$	21112_{L}
$(0, 1211)$	12112_{L}
$(0, 1121)$	11212_{L}
$(0, 221)$	2212_{L}
$(0, 1112)$	11122_{L}
$(0, 212)$	2122_{L}
$(0, 122)$	1222_{L}
$(1, 11111)$	1111111_{L}
$(1, 2111)$	211111_{L}
$(1, 1211)$	121111_{L}
$(1, 1121)$	112111_{L}
$(1, 221)$	22111_{L}
$(1, 1112)$	111211_{L}
$(1, 212)$	21211_{L}
$(1, 122)$	12211_{L}
$(2, 11111)$	111121_{L}
$(2, 2111)$	21121_{L}
$(2, 1211)$	12121_{L}
$(2, 1121)$	11221_{L}
$(2, 221)$	2221_{L}
$(2, 1112)$	111_{R}
$(2, 212)$	21_{R}
$(2, 122)$	12_{R}

Table 2.2: The bijections `bijection (thrice' {4})` and `bijection (thrice' {5})`.

they extended the Fibonacci numbers to negative indices, allowing them to offset the recurrence even further down. There they ended up with what they concluded was the minimum possible size of a base case. There one could choose the base case arbitrarily, and end up with a natural bijection, no matter the choice. In particular, this allowed them to count the number of natural bijections for this identity, and also to classify which base cases for higher offsets were natural. Indeed, the base cases we chose were among those.

But now consider Example 23, where we used our semiring solver, augmented with our heuristic, to automatically prove this identity (shifted to avoid negative indices, so $3F_{n+2} = F_{n+4}+F_n$). We got a different bijection than the one produced in Example 9, although one will find that it is indeed one of the natural bijections classified by Wood and Zeilberger. Yet, the underlying proof produced by the semiring solver is completely different, and, in particular, does not use induction nor any special base cases.

So why is this the case? Let's look at how the solver proves the identity at a high level. First, it applies the heuristic and expands each side of the equation, as follows:

$$3F_{n+2} = 3 \cdot (F_{n+1} + F_n)$$

$$\begin{aligned} F_{n+4} + F_n &= (F_{n+3} + F_{n+2}) + F_n \\ &= ((F_{n+2} + F_{n+1}) + F_{n+2}) + F_n \\ &= (((F_{n+1} + F_n) + F_{n+1}) + (F_{n+1} + F_n)) + F_n \end{aligned}$$

Just as before we introduced the heuristic, the atomic F_{n+1} and F_n will now be treated as variables by the solver. That is, it now goes on to prove the identity

$$3 \cdot (x + y) = (((x + y) + x) + (x + y)) + y$$

It does so by reducing both sides to their normal forms, doing so one step at a time using the various algebraic identities, and ends up with $3 \cdot x + 3 \cdot y$ on both sides, and is thus done.

Let's take an even closer look at the intermediate steps using a simpler identity: $2a + b = a + b + a$. For illustrative purposes, let's begin by expanding the left side of the equation using the identity $2a = a + a$, and thus get[9] $a + a + b = a + b + a$. To prove this identity, the solver would bring both sides of the equation to normal form. We will allow ourselves a slightly simpler variant of the normal form, and simply apply commutativity on the final $b + a$ on the right side of the equation to get our result: $a + a + b = a + a + b$. As these two expressions are syntactically equivalent, it is of course obvious that their values are the same. And thinking in terms of the lifted bijection, an element from the left side of type $A \sqcup A \sqcup B$ can be easily mapped to an element from the right side, also of type $A \sqcup A \sqcup B$, using the identity bijection.

But now notice the following. It is also possible to get a distinct normal form, that is still syntactically equivalent to the expression we had before. If we label the a's, just for illustration, we currently have that $a_1 + a_2 + b = a_3 + a_4 + b$. By applying commutativity on $a_3 + a_4$, we get that $a_1 + a_2 + b = a_4 + a_3 + b$. In terms of the

[9]We allow ourselves to play loosely with associativity, as it does not play a big role in our illustration.

lifted bijection, we can still use the identity bijection at this step, but now there is an additional step changing an `inj₁ a` element of $A \sqcup A$ to an `inj₂ a` element, and vice-versa, thus creating an overall different bijection.

This argument also generalizes to our previous identity: $3 \cdot (x + y) = (((x + y) + x) + (x + y)) + y$. Here we have three x's and three y's, both on the left and right sides of the equation. When in normal form, these six variables will each have to be matched to a variable of the same type on the other side. It is clear that this can be done in $3!^2$ ways, leaving us with $3!^2$ possible bijections for this identity.

One can go on to look at other kinds of identities, such as $4 + 6 = 10$. Expanding both sides of the equation will leave us with a sum of ten 1's on each side of the equation, that then have to be paired up. This can be done in 10! ways, each way giving a distinct bijection. Each 1 in the sum on the left side thus represents a distinct item from the lifted set on the left side of the equality, and similarly each 1 in the sum on the right side represents a distinct item from the lifted set on the right side of the equality. Thus pairing two 1's in the final normal forms means that their corresponding elements will be matched in our final bijection.

Similarly, when we have a larger expression such as F_n, it represents F_n elements from one of the lifted sets. When matching that with another F_n on the other side of the equation, we essentially decide that this bunch of elements should be mapped to that bunch of elements on the other side. One could also decompose F_n into a sum of F_n many 1's, and thus rearrange them for some $F_n!$ possible bijections. This, however, would be a very unnatural way of proving such an identity, and it is this that leaves us with some intuition for when a natural bijection is formed: the atoms in our normal form are few but large, with the elements corresponding to each atom semantically belonging together.

Unfortunately, this classification of a natural bijection does not seem to coincide with the one given by Wood and Zeilberger. In particular, they found that there were 3! natural bijections for the above identity, while we determined that there were $3!^2$ of them. It would be interesting to further compare our definitions of naturality by, for example, inspecting the particular bijections for the above identity, although we have not done so. In any case, this discussion shows how these underlying bijections are constructed from an interesting perspective.

2.7 Cancellation

As we pointed out in Section 1.2, we have not described how to lift expressions involving subtraction or division. Indeed, it is generally not straightforward to assign combinatorial interpretations to such expressions. Nevertheless, Wood and Zeilberger [37] gave a reasonable way to deal with subtraction in the translation method. We will present this technique, albeit in a slightly different manner, as well as extend it with new operations. This is based on an earlier paper by Feldman and Propp [11], not referenced by Wood and Zeilberger.

So far we have seen many identities related to bijections. Consider the following identity from Table 1.2, labeled as "+ congruence": If we have two bijections $f : A \to C$

and $g : B \to D$, then we also have a bijection $h : A \sqcup B \to C \sqcup D$, given by

$$h(x) = \begin{cases} f(a)_{\mathsf{L}} & \text{if } x = a_{\mathsf{L}} \\ g(c)_{\mathsf{R}} & \text{if } x = c_{\mathsf{R}} \end{cases}$$

Of course, if we have knowledge about the specific elements of A, B, C and D, there are a lot more possible bijections between $A \sqcup B$ and $C \sqcup D$ than the above trivial one.

The problem considered by Feldman and Propp was as follows: given a construction as the one described above, where the bijection h can be arbitrary, is it possible to reverse the construction, or *cancel* it? That is, for the case above, given a bijection $g : B \to D$ and a bijection $h : A \sqcup B \to C \sqcup D$, can we recover a bijection $f : A \to C$? And indeed, as this is the inverse of the union of bijections, this can be seen as a form of subtraction for bijections.

They present general criteria for when a construction can be cancelled. They show that this is indeed possible for addition, as shown above, as well as a few other common operations, and give polynomial-time algorithms to perform the respective cancellations. We will add support for some of these operations in our translate module, starting with cancellation of addition.

2.7.1 Cancellation of addition

With the above discussion in mind, we can now present cancellation of addition as the following identity, along with its lifted version:

	Identity	Lifted identity
cancel +	$\dfrac{a + b = c + d \qquad b = d}{a = c}$	$\dfrac{A \sqcup B \cong C \sqcup D \qquad B \cong D}{A \cong C}$

Specified in Agda in terms of our translate module, this has the following type definition:

```
+-cancel : ∀ {a b c d} → a + b ≡ c + d → b ≡ d → a ≡ c
```

Now, to actually implement this function (or alternatively, to prove this property) we will have to provide two things:

- A proof that the identity holds for natural numbers: if $a + b = c + d$ and $b = d$, then $a = c$.
- A bijection from A to C, given the bijections $A \sqcup B \cong C \sqcup D$ and $B \cong D$.

Proving that the identity holds for natural numbers is a simple exercise in Agda. Coming up with the bijection, however, turns out to require some ingenuity.

This is where Feldman and Propp's cancellation procedure comes to play. It is the same procedure Wood and Zeilberger used, and both papers agree that its origins can be traced back to the Involution principle, which was introduced by Garsia and Milne [13].

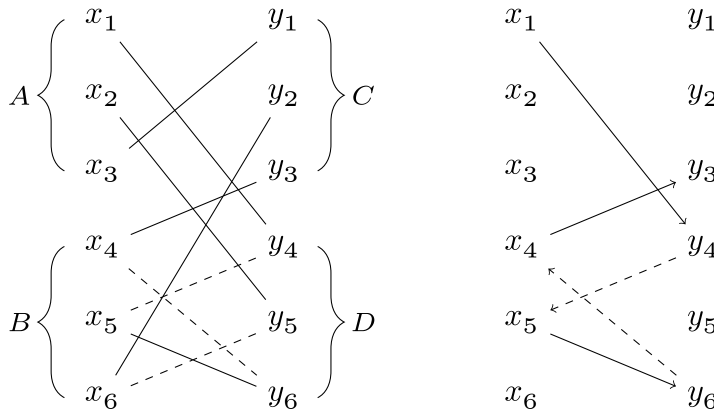

Figure 2.2: An example of the cancellation procedure for addition. On the left we have the bijection $f : A \sqcup B \to C \sqcup D$, denoted with solid line segments, and the bijection $g : B \to D$, denoted with dashed line segments. On the right is a trace of the procedure starting at x_1, which ends at y_3. The resulting bijection $A \cong C$ maps $x_1 \mapsto y_3$, $x_2 \mapsto y_2$ and $x_3 \mapsto y_1$.

The procedure is a mechanical description of the bijection that we require. That is, given an element of A, it uses the bijections $A \sqcup B \cong C \sqcup D$ and $B \cong D$ to produce an element of C. It does so as follows:

Definition 24 (Cancellation of addition)**.** Given an $a \in A$, as well as two bijections $f : A \sqcup B \to C \sqcup D$ and $g : B \to D$, produce an element of C as follows:

1. If $f(a) \in C$, return $f(a)$. Otherwise let $d = f(a)$ and continue:
2. Let $b = g^{-1}(d)$.
3. If $f(b) \in C$, return $f(b)$. Otherwise let $d = f(b)$, and go back to step 2.

Figure 2.2, adapted from [37], gives an example of this procedure.

Assuming that the sets are finite, the fact that this procedure terminates for all inputs $a \in A$ follows from the fact that f and g are bijections: if the procedure doesn't terminate, then it has hit a cycle, in which case either f or g cannot be a bijection. The fact that each $a \in A$ produces a distinct C also follows from the fact that f and g are bijections: simply running the procedure in reverse from the resulting element of C will give us back the same element a.

Unfortunately, implementing this procedure in Agda is not as easy as it sounds. Consider the following Agda pseudocode for the procedure:

```
to : {A B C D : Set} → (A ⊔ B) ≅ (C ⊔ D) → B ≅ D → A ⊔ B → C
to f g x with getTo f x
to f g x | inj₁ y = y
to f g x | inj₂ y = to f g (inj₂ (getFrom g y))
```

While technically a correct implementation of the procedure, it does not pass Agda's termination checker. Furthermore, this is only part of what needs to be implemented to

create a bijection—one would also need to implement the corresponding `from` function, and then prove that they together form a bijection.

Towards that we introduce the following data type to reason about the alternating paths traversed by the procedure:

```
data [_∧_|=_⇒_] {A B C D : Set}
                (f : (A ⊔ B) ≅ (C ⊔ D))
                (g : B ≅ D) : A ⊔ B → C → Set where

  step : ∀ {p q r s}
       → (getTo f p P≡ inj₂ q)
       → (getFrom g q P≡ r)
       → [ f ∧ g |= inj₂ r ⇒ s ]
       → [ f ∧ g |= p ⇒ s ]

  done : ∀ {p q}
       → (getTo f p P≡ inj₁ q)
       → [ f ∧ g |= p ⇒ q ]
```

Specifically, `[ f ∧ g |= x ⇒ y ]` represents a proof that, if we're running the procedure using the bijections f and g, and are currently on the element x (in $A \sqcup C$), then the procedure will produce the element y (in C).

With this data type we can start specifying and proving properties about the procedure as a whole, such as the fact that it's injective:

```
injective : ∀ {A B C D}
              (f : (A ⊔ B) ≅ (C ⊔ D))
              (g : B ≅ D)
              {x y z}
            → [ f ∧ g |= y ⇒ x ]
            → [ f ∧ g |= y ⇒ z ]
            → x P≡ z
```

That is, if we have two alternating paths that start at the same element, then they will both lead to the same element.

Similarly, as we pointed out above, we can specify that, if we start at an element $x \in A \sqcup B$ and end at an element $y \in C$, that if we then start again at y and apply the reverse procedure (i.e. use the inverse bijections), then we will end up at x again.

```
reverse : ∀ {A B C D}
            (f : (A ⊔ B) ≅ (C ⊔ D))
            (g : B ≅ D)
            {x y}
          → [ f ∧ g |= inj₁ x ⇒ y ]
          → [ (Bsym f) ∧ (Bsym g) |= inj₁ y ⇒ x ]
```

Now recall that to prove something is a bijection in our module, one will need to show that `from (to x) P≡ x` (and `to (from y) P≡ y` for the inverse). In terms of our

procedure, we first run the procedure on x to get an element y, and then run the procedure on y using the inverse bijections to get a third element z. We then need to show that $x = z$.

When we run the procedure we can simultaneously construct a proof making use of our new data type. So here we will have two instances, `p` of type `[ f ∧ g |= inj₁ x ⇒ y ]` and `q` of type `[ (Bsym f) ∧ (Bsym g) |= inj₁ y ⇒ z ]`, to work with. These instances, along with the `injective` and `reverse` functions defined above, are enough to prove our `from-to` property:

```
from-to : ∀ x → from (to x) P≡ x
from-to x = injective (Bsym f) (Bsym g) q (reverse f g p)
```

This is the gist of the implementation, but in reality one also needs to apply the fact that `Bsym (Bsym b) P≡ b`.

So now we've proved that the procedure does indeed form a bijection. What remains is then to convince the termination checker that the `to` and `from` functions do terminate.

Our proof above relied on the fact that the four sets are finite, meaning that we will eventually either reach an element in C, or else reach an element in D that we've visited before, contradicting the fact that f and g are bijections. This seems like a nontrivial thing to prove in Agda—and it is—but luckily the Agda standard library comes equipped with a structure to deal specifically with this use case: `Data.List.Countdown`.

The `Countdown` type, denoted with `_⊕_`, keeps track of a set of elements over a given type `T`. Specifically, if `counted` is a list of elements of type `T`, and `n` is a natural number, the type `counted ⊕ n` is a proof that at most `n` elements of type `T` are not in the `counted` list.

The `Countdown` type then supports the following two functions that are of importance to us:

- Given an exhaustive list of elements of type `T`, and a proof that it is indeed exhaustive, construct a `Countdown` type with an empty `counted` list:

  ```
  emptyFromList : (counted : List T)
                → (∀ x → x ∈ counted)
                → [] ⊕ length counted
  ```

- Given an element `x` of type `T`, and a proof that `x` is not in `counted`, add `x` to the `Countdown` list.

  ```
  insert : ∀ {counted n}
         → counted ⊕ suc n
         → ∀ x
         → x ∉ counted
         → x ∷ counted ⊕ n
  ```

The idea, then, is to use two `counted` lists, one for the `A ⊔ B` type and one for the `D` type, to keep track of which elements we have visited on our alternating path. This will help in two ways. First, if we ever visit an element for the second time, then it will already be in one of our `counted` lists. With enough information about the elements

that already exist in our `counted` lists, this is enough to derive a contradiction. Second, as the natural number `n`, the number of uncounted elements in our countdown lists, is strictly decreasing, this will be enough to convince Agda's termination checker that the procedure does indeed terminate.

The precise information we will need to keep track of is as follows:

1. A `counted` list for the `A ⊔ B` type, denoted by `xs`. As `A ⊔ B` is actually a lifted expression, i.e. `lift (a + b)` for some `a` and `b`, we can use the `generate` and `exhaustive` functions introduced in Section 2.1, along with the `emptyFromList` function, to construct this list.

2. A `counted` list for the `D` type, denoted by `ys`. Just as for `xs` we can use the `emptyFromList` function to construct this list. Note that we don't need to keep track of elements from the `C` type, because as soon as we encounter one of them, we're done.

3. For every element x of type `B` in our `xs` list, a proof that there is an element `y` in `ys` that was the predecessor of `x` in our alternating path.

4. Similar to the previous item, for every element `y` in our `ys` list, a proof that there is an element `x` in our `xs` list that is the predecessor of `y` in our alternating path.

5. Finally some information about the current head `x` of our alternating path: that `x` has not been added to the `xs` list, and a proof that there is a `y` in our `ys` list that was the predecessor of `x` in our alternating path.

One may notice a certain cyclic dependency in the information we list above, and that it seems impossible to bootstrap this information. However, the key is that the piece of information in item 3 only needs to be provided for elements of type `B`. In particular, we do not need to provide this piece of information about the unique element of type `A` in our path, namely the first element. We will thus consider it as a special case to bootstrap the procedure.

With the above list of information, the following function is enough to carry out our procedure, except for the first step:

```
run : ∀ {m n}
    → (xs : List (A ⊔ B))
    → xs ⊕ m
    → (ys : List D)
    → ys ⊕ n
    → ((x : A ⊔ B) → x ∈ xs
                   → (inj₂-p : ∃ (λ x' → x P≡ inj₂ x'))
                   → Σ D (λ y → getFrom g y P≡ (proj₁ inj₂-p) × y ∈ ys))
    → ((y' : D) → y' ∈ ys
                → Σ (A ⊔ B) (λ x' → getTo f x' P≡ (inj₂ y') × x' ∈ xs))
    → (x : B)
    → (inj₂ x) ∉ xs
    → Σ D (λ y' → getFrom g y' P≡ x × y' ∈ ys)
    → Σ C (λ y → [ f ∧ g |= inj₂ x ⇒ y ])
```

The implementation of the run function has the same structure as the to function presented in Agda pseudocode above, but with more code to keep track of the required information, as well as to deal with the boundary cases that can come up while running the procedure. In particular, when we encounter an element for the second time.

As we saw above, the current element x is always accompanied with a proof that it is not in its respective counted list. If we then take one step forward in our alternating path to an element y that we have encountered before, i.e. it is in the respective counted list, then we have another element x', which y apparently came from, along with a proof that x' is in its respective counted list. This leads to a contradiction, as proved by the following function:

```
contr : ∀ x x' xs y
      → x ∉ xs
      → x' ∈ xs
      → getTo f x P≡ inj₂ y
      → getTo f x' P≡ inj₂ y
      → ⊥
```

Similarly, a corresponding function gives the contradiction when the repeated element exists in the other counted list. These two functions are used extensively in the run function to deal with contradictory cases that occur.

With the run function implemented, a separate function to bootstrap the procedure, as well as the above proofs that the procedure forms a bijection, this is finally enough to implement cancellation of addition:

```
+-cancel : ∀ {a b c d} → a + b ≡ c + d → b ≡ d → a ≡ c
```

Wood and Zeilberger give one example where cancellation of addition is used, but due to their use of negative indices, the example is not well suited to our translate module. We leave it up to future users of the translate module to find examples where the cancellation of addition is useful.

As a final note we would like to mention that, since Remmel's bijection machine [27] is based on the same alternating paths procedure, one could in principle implement the machine using the translate module and the above cancellation procedure to explore and prove identities involving integer partitions.

2.7.2 Other cancellation procedures

Feldman and Propp also considered operations other than addition, and gave polynomial-time algorithms for those that were cancellable. We will mention their results here, although we have yet to implement cancellation procedures for operations other than addition. Nevertheless we believe that they can be implemented in a similar manner.

Now, after addition, the next operation one may want to consider is multiplication. Unfortunately, Feldman and Propp proved that this operation is not cancellable. That is, given an arbitrary bijection $A \times C \cong B \times C$, it is generally not possible to recover a bijection $A \cong B$. However, they did give a cancellation procedure for the case when C has a distinguished element. For our purposes, this doesn't seem to make much of

a difference, other than to ensure that C is nonempty, as one can practically always pick a distinguished element in C. It should therefore be possible to implement this procedure in our translate module, and we get the following identity:

	Identity	Lifted identity
cancel $\cdot$	$\dfrac{a \cdot c = b \cdot c \qquad c > 0}{a = b}$	$\dfrac{A \times C \cong B \times C \qquad c \in C}{A \cong B}$

Unlike multiplication without a distinguished element, they proved that repeated multiplication, i.e. exponentiation, is cancellable for sets without a distinguished element. That is, given an arbitrary bijection $A^k \cong B^k$, with k a positive integer, it is possible to recover a bijection $A \cong B$. They give a cancellation procedure, which should also be possible to implement in the translate module, and we get the following identity:

	Identity	Lifted identity
cancel ˆ	$\dfrac{a^k = b^k \qquad k > 0}{a = b}$	$\dfrac{A^k \cong B^k \qquad k > 0}{A \cong B}$

Lastly they proved that it is not possible to cancel exponentials. That is, given an arbitrary bijection $2^A \cong 2^B$, it is generally not possible to recover a bijection $A \cong B$. This result is a bit unfortunate, as it eliminates some possible—although far-fetched, for now—applications of the translation method.

2.8 Further examples

Before concluding, let us now present two final examples.

Example 25. Consider problem 108 from Stanley's Enumerative combinatorics [32]:

> Give a combinatorial proof that the number of partitions of $[n]$ such that no two consecutive integers appear in the same block is the Bell number $B(n-1)$.

Let's solve this problem using the translation method and the translate module. First recall that the number of partitions of $[n]$ are given by the Bell number $B(n)$. Furthermore,

$$B(n) = \sum_{k=0}^{n} S(n,k),$$

where $S(n,k)$, the Stirling number of the second kind, counts the number of partitions of $[n]$ containing exactly k blocks. Now take any such partition and remove the largest element, n. Two cases may occur:

- The element n was in a singleton block, in which case the remaining blocks form a partition of $[n-1]$ with $k-1$ blocks.
- The element n was in a block with other elements, in which case we now have a partition of $[n-1]$ that still has k blocks. This can happen in k different ways depending on which block the element n was contained in.

This gives rise to the following recurrence relation for the Stirling numbers of the second kind:

$$S(n,k) = k \cdot S(n-1,k) + S(n-1,k-1)$$

if $k > 0$, with the base cases $S(0,0) = 1$ and $S(n,0) = S(0,n) = 0$.

Now let $B'(n)$ denote the number of partitions of $[n]$ such that no two consecutive integers appear in the same block, and $S'(n,k)$ denote the number of such partitions with exactly k blocks. As before we have that

$$B'(n) = \sum_{k=0}^{n} S'(n,k).$$

Let's again take any such partition on n elements containing k blocks, and remove the largest element, n. Again we have two cases:

- The element n was in a singleton block, in which case the remaining blocks form such a partition of $[n-1]$ with $k-1$ blocks.
- The element n was in a block with other elements, in which case we now have such a partition of $[n-1]$ that still has k blocks. This can happen in $k-1$ different ways depending on which block the element n was contained in, which can be any of the k blocks except for the one that contains the element $n-1$.

This gives the following recurrence relation for our modified Stirling numbers of the second kind:

$$S'(n,k) = (k-1) \cdot S'(n-1,k) + S'(n-1,k-1)$$

if $k > 0$, with the base cases $S'(0,0) = 1$ and $S'(n,0) = S'(0,n) = 0$.

The problem asks us to give a combinatorial proof that $B'(n) = B(n-1)$. However, after some experimentation, one may find that a stronger result holds: $S'(n,k) = S(n-1,k-1)$. This can be proved by induction. When $n = k = 1$, then $S'(n,k) = 1 = S(n-1,k-1)$. When $n = 1$ and $k > 1$, then $S'(n,k) = 0 = S(n-1,k-1)$. Finally, for $n > 1$ and $k > 1$,

$$\begin{aligned} S'(n,k) &= (k-1) \cdot S'(n-1,k) + S'(n-1,k-1) && \text{definition} \\ &= (k-1) \cdot S(n-2,k-1) + S(n-2,k-2) && \text{induction} \\ &= S(n-1,k-1) && \text{definition} \end{aligned}$$

Now, by applying the translation method to this algebraic proof, we might be able to find a bijection between partitions of $[n]$ with k blocks that have no two consecutive integers, and the partitions of $[n-1]$ with $k-1$ blocks. Unfortunately there is currently no support for working with the Bell and Stirling numbers in the translate module, so we will use this opportunity to show how one would add support for such functions.

First we will need some basic types and functions to work with. In the translate module source directory, these will go into the file `Translate/Types.agda`. Let's start by implementing our recurrence from above for the Stirling numbers of the second kind:

```
ℕS : (n k : ℕ) → ℕ
ℕS ℕzero ℕzero = 1
ℕS ℕzero (ℕsuc k) = 0
ℕS (ℕsuc n) ℕzero = 0
ℕS (ℕsuc n) (ℕsuc k) = (ℕsuc k) ℕ* ℕS n (ℕsuc k) ℕ+ ℕS n k
```

The function for the modified Stirling numbers, `ℕS'`, is implemented similarly.

Next we will need data types to represent the lifted types. In our case they are the set partitions on $[n]$ with k blocks, as well as the modified version without consecutive integers. Our representation will be exactly as we described when we gave the recurrence for the Stirling numbers:

```
data SetPartitionK : ℕ → ℕ → Set where
  empty : SetPartitionK ℕzero ℕzero
  add : ∀ {n k} → SetPartitionK n k → SetPartitionK (ℕsuc n) (ℕsuc k)
  insert : ∀ {n k} → Fin k → SetPartitionK n k → SetPartitionK (ℕsuc n) k
```

Here `empty` represents the unique partition of the empty set, `add` represents a partition where the largest element, n, is in a singleton block, and `insert` represents a partition where the largest element is put in one of the k blocks of a smaller partition on $n-1$ elements. The modified version will also be similar, but with $k-1$ choices instead of k.

Next we will need to implement each of the functions that we introduced in Section 2.1:

```
show : ∀ {n k} → SetPartitionK n k → String

generate : ∀ {n k} → List (SetPartitionK n k)

exhaustive : ∀ {n k} → (x : SetPartitionK n k) → x ∈ generate {n} {k}

equal : ∀ {n k} → (x y : SetPartitionK n k) → Dec (x P≡ y)
```

The `generate`, `exhaustive` and `equal` functions are pretty standard, but we have some freedom in how we implement the `show` function. Given an instance of the `SetPartitionK` data type, one could use that to build a more conventional representation of a set partition, and use a textual representation for the `show` function. However, we decided to give a more direct representation: `empty` is represented as the empty string, `add p` is represented as 0, and `insert i p` is represented as $i+1$. These are then strung together to form our representation for the `show` function. For example, the set partition `insert Fzero (insert Fzero (add empty))` is represented as `011`.

Finally we need to incorporate these new types and functions into our framework. To do that, we will first update our `Expr` data type with two new constructors:

```
S : (n k : ℕ) → Expr
S' : (n k : ℕ) → Expr
```

Type checking the Agda file will now fail, pointing to a couple of functions that are missing cases for our newly added constructors. Implementing the relevant cases,

which should be a simple matter of adding the case and pointing to our new version of `show`, `generate`, `exhaustive` or `equal`, fixes the errors.

Next we will update the file `Translate/Combinatorics.agda`, where combinatorial identities belong. In our case, we will prove the following four defining identities for our Stirling numbers:

```
S-def₁ : ∀ {n k}
       → S (ℕsuc n) (ℕsuc k) ≡ (nat (ℕsuc k)) * S n (ℕsuc k) + S n k

S-def₂ : ∀ {n} → S (ℕsuc n) ℕzero ≡ zero

S'-def₁ : ∀ {n k}
        → S' (ℕsuc n) (ℕsuc k) ≡ (nat k) * S' n (ℕsuc k) + S' n k

S'-def₂ : ∀ {n} → S' (ℕsuc (ℕsuc n)) (ℕsuc ℕzero) ≡ zero
```

This is all we have to do to add support for our functions in the translate module. To add solver support for our functions, one still needs to do some extra work in `Translate/Solver.agda` and `Translate/Solver/Types.agda`. While straightforward, we won't be using the solver here, so we will skip that part for now.

We can now finally implement our above proof in Agda:

```
part : ∀ {n k} → S' (ℕsuc n) (ℕsuc k) ≡ S n k

-- Base case: S'(1, 1) = S(0, 0)
part {ℕzero} {ℕzero} = proof Prefl (mkBij to from to-from from-to)
  where
    to : lift (S' (ℕsuc ℕzero) (ℕsuc ℕzero)) → lift (S ℕzero ℕzero)
    to (add empty) = empty
    to (insert () _)

    from : lift (S ℕzero ℕzero) → lift (S' (ℕsuc ℕzero) (ℕsuc ℕzero))
    from empty = add empty

    to-from : ∀ x → to (from x) P≡ x
    to-from empty = Prefl

    from-to : ∀ y → from (to y) P≡ y
    from-to (add empty) = Prefl
    from-to (insert () _)

-- Base case: S'(1, 2+k) = S(0, 1+k)
part {ℕzero} {ℕsuc k} = proof (Ptrans (NPS.+-right-identity (k ℕ* 0))
                                      (NPS.*-right-zero k))
                              (mkBij to from to-from from-to)
  where
    to : lift (S' (ℕsuc ℕzero) (ℕsuc (ℕsuc k))) → lift (S ℕzero (ℕsuc k))
    to (add ())
```

```
    to (insert _ ())

    from : lift (S ℕzero (ℕsuc k)) → lift (S' (ℕsuc ℕzero) (ℕsuc (ℕsuc k)))
    from ()

    to-from : ∀ x → to (from x) P≡ x
    to-from ()

    from-to : ∀ y → from (to y) P≡ y
    from-to (add ())
    from-to (insert _ ())

-- Inductive case: S'(2+n, 1) = S(1+n, 0)
part {ℕsuc n} {ℕzero} =
  begin
    S' (ℕsuc (ℕsuc n)) (ℕsuc ℕzero)
  ≡⟨ S'-def₂ ⟩
    zero
  ≡⟨ sym S-def₂ ⟩
    S (ℕsuc n) ℕzero
  ∎

-- Inductive case: S'(2+n, 2+k) = S(1+n, 1+k)
part {ℕsuc n} {ℕsuc k} =
  begin
    S' (ℕsuc (ℕsuc n)) (ℕsuc (ℕsuc k))
  ≡⟨ S'-def₁ ⟩
    (nat (ℕsuc k)) * S' (ℕsuc n) (ℕsuc (ℕsuc k)) + S' (ℕsuc n) (ℕsuc k)
  ≡⟨ +-cong (*-cong refl part) part ⟩
    (nat (ℕsuc k)) * S n (ℕsuc k) + S n k
  ≡⟨ sym S-def₁ ⟩
    S (ℕsuc n) (ℕsuc k)
  ∎
```

We can now apply the translation method by simply calling the `bijection` function on our identity. The resulting bijection is partially given in Table 2.3.

By looking at this table, as well as further outputs given by the bijection, one can easily guess that the direct bijection simply drops the first 0 of the input. The fact that this forms a bijection is straightforward to prove, and is left as an exercise for the reader.

This example illustrates that sometimes just the act of formalizing the problem for the translate module goes a long way towards finding a bijection. In this case the most important part was how we chose our representation of the set partitions. We will also note that the results of this example easily generalize to the identity $B'_d(n) = B(n-d)$, where $B'_d(n)$ denotes the number of partitions of $[n]$ where all pairs of elements in a block have absolute difference greater than d. (See [26])

Example 26. As a final example let's consider Cassini's identity, $F_{n+1}F_{n-1} = F_n^2 + (-1)^n$ for $n \geq 1$, which was also given by Wood and Zeilberger. First, to get rid of

$S(n,k)$	$S'(n-1,k-1)$
00001	0001
00002	0002
00003	0003
00010	0010
00020	0020
00100	0100
000111	00111
000211	00211
001011	01011
000121	00121
000221	00221
001021	01021
001101	01101
000112	00112
000212	00212
001012	01012
000122	00122
000222	00222
001022	01022
001102	01102
001110	01110

Table 2.3: A few samples from `bijection (part n k)`, using the textual representation given by `show`.

the negative number, we will split this identity into two cases, depending on if n is even or odd:

$$F_{2k+2}F_{2k} + 1 = F_{2k+1}^2$$
$$F_{2k+3}F_{2k+1} = F_{2k+2}^2 + 1$$

Instead of proving these identities on paper, we will go straight to the Agda implementation. Indeed, we usually prefer to go straight to Agda, using the facilities it provides[10] to develop proofs.

```
cassini-odd : ∀ k → fib (2 ℕ* k ℕ+ 2) * fib (2 ℕ* k)
                  ≡ fib (2 ℕ* k ℕ+ 1) * fib (2 ℕ* k ℕ+ 1) + one
cassini-odd ℕzero = proof Prefl (mkBij to from to-from from-to)
  where
    to : (FibStr 2 × FibStr 0) → (FibStr 1 × FibStr 1 ⊔ Maybe ⊥)
    to (([] ::1 ::1) , []) = inj₁ (([] ::1) , ([] ::1))
    to (([] ::2) , []) = inj₂ nothing

    from : (FibStr 1 × FibStr 1 ⊔ Maybe ⊥) → (FibStr 2 × FibStr 0)
    from (inj₂ nothing) = ([] ::2) , []
    from (inj₁ (([] ::1) , ([] ::1))) = ([] ::1 ::1) , []
    from (inj₂ (just ()))

    to-from : ∀ y → to (from y) P≡ y
    to-from (inj₁ (([] ::1) , ([] ::1))) = Prefl
    to-from (inj₂ (just ()))
    to-from (inj₂ nothing) = Prefl

    from-to : ∀ x → from (to x) P≡ x
    from-to ((([] ::1) ::1) , []) = Prefl
    from-to (([] ::2) , []) = Prefl

cassini-odd (ℕsuc k) =
  begin
    fib (2 ℕ* (ℕsuc k) ℕ+ 2) * fib (2 ℕ* (ℕsuc k))
  P≡⟨ Pcong (λ x → fib x * fib (2 ℕ* (ℕsuc k))) (NPS.+-comm (2 ℕ* (ℕsuc k)) 2) ⟩
    (fib (ℕsuc (ℕsuc (2 ℕ* (ℕsuc k))))) * fib (2 ℕ* (ℕsuc k))
  ≡⟨ *-cong fib-def refl ⟩
    (fib (ℕsuc (2 ℕ* (ℕsuc k))) + fib (2 ℕ* (ℕsuc k))) * fib (2 ℕ* (ℕsuc k))
  ≡⟨ distribʳ-*-+ ⟩
    fib (ℕsuc (2 ℕ* (ℕsuc k))) * fib (2 ℕ* (ℕsuc k)) + fib (2 ℕ* (ℕsuc k)) * fib (2 ℕ* (ℕsuc k))
  P≡⟨ Pcong (λ x → fib (ℕsuc (2 ℕ* (ℕsuc k))) * fib (2 ℕ* (ℕsuc k)) + fib x * fib (2 ℕ* (ℕsuc k)))
            (lem2 2 k) ⟩
    fib (ℕsuc (2 ℕ* (ℕsuc k))) * fib (2 ℕ* (ℕsuc k)) + fib (2 ℕ* k ℕ+ 2) * fib (2 ℕ* (ℕsuc k))
  P≡⟨ Pcong (λ x → fib (ℕsuc (2 ℕ* (ℕsuc k))) * fib (2 ℕ* (ℕsuc k)) + fib (2 ℕ* k ℕ+ 2) * fib x)
            (lem2 2 k) ⟩
    fib (ℕsuc (2 ℕ* (ℕsuc k))) * fib (2 ℕ* (ℕsuc k)) + fib (2 ℕ* k ℕ+ 2) * fib (2 ℕ* k ℕ+ 2)
  ≡⟨ +-cong refl (cassini-even k) ⟩
    fib (ℕsuc (2 ℕ* (ℕsuc k))) * fib (2 ℕ* (ℕsuc k)) + (fib (2 ℕ* k ℕ+ 3) * fib (2 ℕ* k ℕ+ 1) + one)
  ≡⟨ sym +-assoc ⟩
    (fib (ℕsuc (2 ℕ* (ℕsuc k))) * fib (2 ℕ* (ℕsuc k)) + fib (2 ℕ* k ℕ+ 3) * fib (2 ℕ* k ℕ+ 1)) + one
  P≡⟨ Pcong (λ x → (fib x * fib (2 ℕ* (ℕsuc k)) + fib (2 ℕ* k ℕ+ 3) * fib (2 ℕ* k ℕ+ 1)) + one)
            (NPS.+-comm 1 (2 ℕ* (ℕsuc k))) ⟩
    (fib ((2 ℕ* (ℕsuc k)) ℕ+ 1) * fib (2 ℕ* (ℕsuc k)) + fib (2 ℕ* k ℕ+ 3) * fib (2 ℕ* k ℕ+ 1)) + one
```

[10] As previously mentioned, "holes" are a very useful feature when writing Agda code, especially proofs. See e.g. [33].

```
  P≡⟨ Pcong (λ x → (fib (x ℕ+ 1) * fib (2 ℕ* (ℕsuc k)) + fib (2 ℕ* k ℕ+ 3) * fib (2 ℕ* k ℕ+ 1)) + one)
          (lem2 2 k) ⟩
    (fib ((2 ℕ* k ℕ+ 2) ℕ+ 1) * fib (2 ℕ* (ℕsuc k)) + fib (2 ℕ* k ℕ+ 3) * fib (2 ℕ* k ℕ+ 1)) + one
  P≡⟨ Pcong (λ x → (fib x * fib (2 ℕ* (ℕsuc k)) + fib (2 ℕ* k ℕ+ 3) * fib (2 ℕ* k ℕ+ 1)) + one)
          (NPS.+-assoc (2 ℕ* k) 2 1) ⟩
    (fib (2 ℕ* k ℕ+ 3) * fib (2 ℕ* (ℕsuc k)) + fib (2 ℕ* k ℕ+ 3) * fib (2 ℕ* k ℕ+ 1)) + one
  ≡⟨ +-cong (sym distribˡ-*-+) refl ⟩
    fib (2 ℕ* k ℕ+ 3) * (fib (2 ℕ* (ℕsuc k)) + fib (2 ℕ* k ℕ+ 1)) + one
  P≡⟨ Pcong (λ x → fib (2 ℕ* k ℕ+ 3) * (fib x + fib (2 ℕ* k ℕ+ 1)) + one) (lem2 2 k) ⟩
    fib (2 ℕ* k ℕ+ 3) * (fib (2 ℕ* k ℕ+ 2) + fib (2 ℕ* k ℕ+ 1)) + one
  P≡⟨ Pcong (λ x → fib (2 ℕ* k ℕ+ 3) * (fib x + fib (2 ℕ* k ℕ+ 1)) + one) (NPS.+-suc (2 ℕ* k) 1) ⟩
    fib (2 ℕ* k ℕ+ 3) * (fib (ℕsuc (2 ℕ* k ℕ+ 1)) + fib (2 ℕ* k ℕ+ 1)) + one
  ≡⟨ +-cong (*-cong refl (sym fib-def)) refl ⟩
    fib (2 ℕ* k ℕ+ 3) * (fib (ℕsuc (ℕsuc (2 ℕ* k ℕ+ 1)))) + one
  P≡⟨ Pcong (λ x → fib (2 ℕ* k ℕ+ 3) * (fib x) + one) (NPS.+-comm 2 (2 ℕ* k ℕ+ 1)) ⟩
    fib (2 ℕ* k ℕ+ 3) * (fib ((2 ℕ* k ℕ+ 1) ℕ+ 2)) + one
  P≡⟨ Pcong (λ x → fib (2 ℕ* k ℕ+ 3) * (fib x) + one) (NPS.+-assoc (2 ℕ* k) 1 2) ⟩
    fib (2 ℕ* k ℕ+ 3) * fib (2 ℕ* k ℕ+ 3) + one
  P≡⟨ Pcong (λ x → fib x * fib (2 ℕ* k ℕ+ 3) + one) (Psym (NPS.+-assoc (2 ℕ* k) 2 1)) ⟩
    fib ((2 ℕ* k ℕ+ 2) ℕ+ 1) * fib (2 ℕ* k ℕ+ 3) + one
  P≡⟨ Pcong (λ x → fib (x ℕ+ 1) * fib (2 ℕ* k ℕ+ 3) + one) (Psym (lem2 2 k)) ⟩
    fib ((2 ℕ* (ℕsuc k)) ℕ+ 1) * fib (2 ℕ* k ℕ+ 3) + one
  P≡⟨ Pcong (λ x → fib ((2 ℕ* (ℕsuc k)) ℕ+ 1) * fib x + one) (Psym (NPS.+-assoc (2 ℕ* k) 2 1)) ⟩
    fib ((2 ℕ* (ℕsuc k)) ℕ+ 1) * fib ((2 ℕ* k ℕ+ 2) ℕ+ 1) + one
  P≡⟨ Pcong (λ x → fib ((2 ℕ* (ℕsuc k)) ℕ+ 1) * fib (x ℕ+ 1) + one) (Psym (lem2 2 k)) ⟩
    fib (2 ℕ* (ℕsuc k) ℕ+ 1) * fib (2 ℕ* (ℕsuc k) ℕ+ 1) + one
  ∎

cassini-even : ∀ k → fib (2 ℕ* k ℕ+ 2) * fib (2 ℕ* k ℕ+ 2)
                   ≡ fib (2 ℕ* k ℕ+ 3) * fib (2 ℕ* k ℕ+ 1) + one
cassini-even k =
  begin
    fib (2 ℕ* k ℕ+ 2) * fib (2 ℕ* k ℕ+ 2)
  P≡⟨ Pcong (λ x → fib (2 ℕ* k ℕ+ 2) * fib x) (NPS.+-comm (2 ℕ* k) 2) ⟩
    fib (2 ℕ* k ℕ+ 2) * fib (ℕsuc (ℕsuc (2 ℕ* k)))
  ≡⟨ *-cong refl fib-def ⟩
    fib (2 ℕ* k ℕ+ 2) * (fib (ℕsuc (2 ℕ* k)) + fib (2 ℕ* k))
  ≡⟨ distribˡ-*-+ ⟩
    fib (2 ℕ* k ℕ+ 2) * fib (ℕsuc (2 ℕ* k)) + fib (2 ℕ* k ℕ+ 2) * fib (2 ℕ* k)
  ≡⟨ +-cong refl (cassini-odd k) ⟩
    fib (2 ℕ* k ℕ+ 2) * fib (ℕsuc (2 ℕ* k)) + (fib (2 ℕ* k ℕ+ 1) * fib (2 ℕ* k ℕ+ 1) + one)
  ≡⟨ sym +-assoc ⟩
    (fib (2 ℕ* k ℕ+ 2) * fib (ℕsuc (2 ℕ* k)) + fib (2 ℕ* k ℕ+ 1) * fib (2 ℕ* k ℕ+ 1)) + one
  P≡⟨ Pcong (λ x → (fib (2 ℕ* k ℕ+ 2) * fib x + fib (2 ℕ* k ℕ+ 1) * fib (2 ℕ* k ℕ+ 1)) + one)
          (NPS.+-comm 1 (2 ℕ* k)) ⟩
    (fib (2 ℕ* k ℕ+ 2) * fib (2 ℕ* k ℕ+ 1) + fib (2 ℕ* k ℕ+ 1) * fib (2 ℕ* k ℕ+ 1)) + one
  ≡⟨ +-cong (sym distribʳ-*-+) refl ⟩
    (fib (2 ℕ* k ℕ+ 2) + fib (2 ℕ* k ℕ+ 1)) * fib (2 ℕ* k ℕ+ 1) + one
  P≡⟨ Pcong (λ x → (fib x + fib (2 ℕ* k ℕ+ 1)) * fib (2 ℕ* k ℕ+ 1) + one) (NPS.+-suc (2 ℕ* k) 1) ⟩
    (fib (ℕsuc (2 ℕ* k ℕ+ 1)) + fib (2 ℕ* k ℕ+ 1)) * fib (2 ℕ* k ℕ+ 1) + one
  ≡⟨ +-cong (*-cong (sym fib-def) refl) refl ⟩
    fib (ℕsuc (ℕsuc (2 ℕ* k ℕ+ 1))) * fib (2 ℕ* k ℕ+ 1) + one
  P≡⟨ Pcong (λ x → fib x * fib (2 ℕ* k ℕ+ 1) + one) (NPS.+-comm 2 (k ℕ+ (k ℕ+ ℕzero) ℕ+ 1)) ⟩
    fib ((2 ℕ* k ℕ+ 1) ℕ+ 2) * fib (2 ℕ* k ℕ+ 1) + one
  P≡⟨ Pcong (λ x → fib x * fib (2 ℕ* k ℕ+ 1) + one) (NPS.+-assoc (2 ℕ* k) 1 2) ⟩
    fib (2 ℕ* k ℕ+ 3) * fib (2 ℕ* k ℕ+ 1) + one
  ∎
```

First, notice the base case given for the odd identity. There are only two possible bijections, so one can try both of them to see which one produces a more natural final bijection. With the base cases above, calling `bijection` on these identities gives us the same bijection that Wood and Zeilberger found. The direct bijection turns out to be very elegant; we refer the interested reader to their paper, [37], for more information.

The proofs above are quite long and tedious. However, notice that most of the proof steps are basic algebra. Although one will not be able to use the solver for the whole proof, it should be possible to use the solver for all steps except those that make use of the inductive hypothesis. That should drastically cut down the length of the proof. We leave it as future work to develop a hybrid version of the equational reasoning environment that allows one to use the solver for some of the steps.

Chapter 3

Conclusions and future work

In this thesis we formalized Wood and Zeilberger's translation method, and gave an implementation of it, known as the translate module, in the programming language Agda. The module is open software, and its source code can be found on GitHub [16]. This formalized version of the translation method is, in our opinion, both more intuitive to apply, as well as more robust. In particular, thanks to Agda's powerful type system, the output produced by the translate module will always be a valid bijection between the two given sets, and provably so.

The translate module already comes with a small library of identities ready for use, and we showed how some of them were implemented. We also adapted Agda's equational reasoning environment to our translate module, making it very natural to specify algebraic proofs, perhaps using the identities from the small library as building blocks. Applying the translation method is then just a simple call to the `bijection` function.

We also showed how, in our formalization, expressions form a commutative semiring. This opens up doors for many interesting uses of the translation method. As an example of that, we implemented a semiring solver that can, in many cases, automatically prove a given identity, and simultaneously apply the translation method. This works for all identities on polynomials. We additionally introduced a heuristic that makes the solver even more powerful. In a sense, this can be thought of as a first step towards making the translation method a fully automatic method.

We looked at the translation method from a philosophical standpoint, investigating how each element is transported through the algebraic proof to form the final bijection. This also gave us some intuition for what a natural bijection is, allowing us to count such bijections for an identity already considered by Wood and Zeilberger, and to compare our meaning of a natural bijection with theirs.

To allow for the use of subtraction in our translate module, we implemented one of Feldman and Propp's cancellation procedures. This was nontrivial to do in Agda, as we had to assure Agda that the procedure terminated, as well as show that the resulting mapping formed a bijection. While Wood and Zeilberger had already introduced support for subtraction in the translation method, we also showed how one might implement other operations, such as division and taking the k-th root, by using Feldman and Propp's procedures.

Although not a big focus of the thesis, we concluded with some applications of the translation method, using our translate module.

We felt that our choice of Agda as an implementation language was a good one; as a hybrid between a theorem prover and an ordinary functional programming language, it fits very well with the translation method where we both work with formal algebraic proofs as well as with concrete bijections. While the formal part of Agda does make it a bit harder to implement some components of the translate module, such as the cancellation procedures, it is also very comforting to know that these components are logically sound, and are thus guaranteed to work as advertised.

While we believe that the translate module is ready for use by combinatorialists, there are still many things that can be improved. One of our goals was to make proofs with the translate module look as similar as possible to ordinary algebraic proofs in Agda. While we think we achieved that to a great extent, there are still some things that stick out like a sore thumb. In particular, if one wants to specify a natural number literal, it has to be prefixed with the term `nat`, e.g. 3 is written as `nat 3`. Similarly, when one wants to use Agda's natural numbers, one has to prefix the relevant terms with ℕ. While it seems that the first issue is not easily fixed without resulting to modifying Agda itself, it may be possible to fix the second issue by switching to the `Expr` data type entirely, instead of using Agda's natural numbers. Unfortunately this may have other downsides.

Even though it is currently straightforward to use the semiring solver, it might be possible to make it seamless by merging the `Expr` and `:Expr` data types. This could also be incorporated into the equational reasoning environment, allowing one to skip steps that are easily proved by the semiring solver.

One thing that Wood and Zeilberger do extensively when using the translation method is to use small indices for their base cases, generalizing down to negative indices if possible, to make the base cases as small as possible. This is currently not straightforward to do in the translate module, as one cannot easily represent negative numbers. Since the number of bijections grows exponentially with the size of the base case, it becomes harder and harder to find a natural base case as they become bigger. This is, therefore, something that would be very valuable to fix in the translate module.

Adding support for new data types to work with using the translate module currently requires one to directly modify the source code of the module, as well as to define multiple functions that are used by the module internally. While we did find ways to allow new data types to be defined externally while designing the translate module, we did not find a design that both allowed that as well as made proofs in the translate module look similar to the ordinary algebraic proofs in Agda. It might be possible to fix the latter issue, however, by using tools like reflection to implement the relevant functions automatically. In particular, reflection has already been used to automatically implement the functionality required by our `equal` function [22].

It still remains to implement the rest of the cancellation procedures given by Feldman and Propp. While nontrivial, we believe they can be implemented in a manner similar to the cancellation procedure for addition.

One last feature that we would like to introduce for the translate module is that of delaying definition of base cases. That is, it should be possible to prove identities in the translate module without specifying concrete base cases. This would then produce a bijection parameterized by the necessary base cases, allowing combinatorialists to

see a partial version of the bijection, as well as giving them the ability to experiment with different base cases in order to discover a natural bijection.

As for the translation method itself, we believe there is a lot of room for extensions to make it more powerful. It would be interesting to see if the theory of combinatorial species could be incorporated into the translation method, and, towards that, functional equations [3]. At the very least it should be possible to make use of its power to assign combinatorial meaning to expressions and operations, such as the derivative [1, 20]. Also, in a way similar to how virtual species are introduced, it might be interesting to see if one can apply the Grothendieck construction to give a meaning to negative numbers in the translation method.

In categorification, sets are replaced by categories [2]. This can be seen as a category-theoretic version of lifting, where algebraic expressions are replaced by sets. Similarly, by means of categorification, it would be interesting to see if there exists a category-theoretic analogue of the translation method.

Finally, there are many different ways to prove algebraic identities, and the viability of applying the translation method to these proof methods should be investigated. One example—perhaps far-fetched—is that of Wilf-Zeilberger pairs [25].

Bibliography

[1] M. Abbott, T. Altenkirch, C. McBride, and N. Ghani, "∂ for data: differentiating data structures", *Fundamenta Informaticae*, vol. 65, no. 1-2, pp. 1–28, 2005.

[2] J. C. Baez and J. Dolan, "Categorification", *arXiv preprint math/9802029*, 1998.

[3] F. Bergeron, G. Labelle, and P. Leroux, *Combinatorial species and tree-like structures*, 67. Cambridge University Press, 1998.

[4] A. Blass, "Seven trees in one", *Journal of Pure and Applied Algebra*, vol. 103, no. 1, pp. 1–21, 1995.

[5] E. Brady, *Type-Driven Development with Idris*. Manning Publications, Mar. 2017, ISBN: 9781617293023.

[6] E. Brady *et al.*, "Idris, a language with dependent types", *University of St. Andrews, http://www.idris-lang.org*, 2009.

[7] T. Y. Chow, *Automated bijective proofs*, https://www.cs.nyu.edu/pipermail/fom/2004-June/008285.html.

[8] D. Christiansen and E. C. Brady, "Elaborator reflection: extending Idris in Idris", in *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ACM Press-Association for Computing Machinery, 2016.

[9] S. Dolan, "Fun with semirings: a functional pearl on the abuse of linear algebra", in *ACM SIGPLAN Notices*, ACM, vol. 48, 2013, pp. 101–110.

[10] J. Esparza, M. Luttenberger, and M. Schlund, "FPsolve: A generic solver for fixpoint equations over semirings", *International Journal of Foundations of Computer Science*, vol. 26, no. 07, pp. 805–825, 2015.

[11] D. Feldman and J. Propp, "Producing new bijections from old", *Advances in Mathematics*, vol. 113, no. 1, pp. 1–44, 1995.

[12] M. Fiore and T. Leinster, "Objects of categories as complex numbers", *Advances in Mathematics*, vol. 190, no. 2, pp. 264–277, 2005.

[13] A. M. Garsia and S. C. Milne, "Method for constructing bijections for classical partition identities", *Proceedings of the National Academy of Sciences*, vol. 78, no. 4, pp. 2026–2028, 1981.

[14] D. Gerdemann, "Combinatorial proofs of Zeckendorf family identities", *Fibonacci Quarterly*, vol. 46, no. 47, p. 2009, 2008.

[15] B. Grégoire and A. Mahboubi, "Proving equalities in a commutative ring done right in Coq", in *International Conference on Theorem Proving in Higher Order Logics*, Springer, 2005, pp. 98–113.

[16] B. A. Guðmundsson, *The translate module for Agda*, https://github.com/SuprDewd/agda-translation-method.

[17] G. Huet, T. Coquand, *et al.*, *The Coq proof assistant*, `https://coq.inria.fr/`.

[18] D. J. Lehmann, "Algebraic structures for transitive closure", *Theoretical Computer Science*, vol. 4, no. 1, pp. 59–76, 1977.

[19] P. Martin-Löf and G. Sambin, *Intuitionistic type theory*. Bibliopolis Napoli, 1984, vol. 9.

[20] C. McBride, "The derivative of a regular type is its type of one-hole contexts", *Unpublished manuscript*, pp. 74–88, 2001.

[21] U. Norell and N. Danielsson, *The agda wiki*, 2005.

[22] U. Norell, *Decidable equality using reflection in Agda*, `https://github.com/UlfNorell/agda-prelude/blob/master/src/Tactic/Deriving/Eq.agda`.

[23] ——, "Dependently typed programming in Agda", in *Advanced Functional Programming*, Springer, 2009, pp. 230–266.

[24] U. Norell *et al.*, *Agda changelog*, `https://github.com/agda/agda/blob/master/CHANGELOG.md`.

[25] M. Petkovsek, H. Wilf, and D. Zeilberger, *A=B*. AK Peters, Ltd., 1996.

[26] H. Prodinger, "On the number of Fibonacci partitions of a set", *Fibonacci Quarterly*, vol. 19, no. 5, pp. 463–465, 1981.

[27] J. B. Remmel, "Bijective proofs of some classical partition identities", *Journal of Combinatorial Theory, Series A*, vol. 33, no. 3, pp. 273–286, 1982.

[28] N. Shar, "Bijective Proofs of Vajda's Ninetieth Fibonacci Number Identity and Related Identities", *Integers*, vol. 12, no. 4, pp. 491–501, 2012.

[29] ——, "Computer-assisted bijectification of Franel's recurrence", *Journal of Difference Equations and Applications*, vol. 20, no. 12, pp. 1583–1591, 2014.

[30] ——, "Experimental methods in permutation patterns and bijective proof", PhD thesis, Rutgers, The State University of New Jersey, 2016.

[31] ——, "Transforming Inductive Proofs to Bijective Proofs", Master's thesis, Stanford University, 2010.

[32] R. P. Stanley, "Enumerative combinatorics, Wadsworth Publ", *Co., Belmont, CA*, 1986.

[33] A. Stump, *Verified functional programming in Agda*. Morgan & Claypool, 2016.

[34] A. M. Turing, "On computable numbers, with an application to the Entscheidungsproblem", *Proceedings of the London mathematical society*, vol. 2, no. 1, pp. 230–265, 1937.

[35] V. Voevodsky *et al.*, "Homotopy type theory: Univalent foundations of mathematics", *Institute for Advanced Study (Princeton), The Univalent Foundations Program*, 2013.

[36] P. van der Walt, "Reflection in Agda", Master's thesis, 2012.

[37] P. M. Wood and D. Zeilberger, "A translation method for finding combinatorial bijections", *Annals of Combinatorics*, vol. 13, no. 3, p. 383, 2009.

[38] N. Zeilberger, N. Pouillard, and N. A. Danielsson, *RingSolver in Agda*, `https://github.com/agda/agda-stdlib/blob/master/src/Algebra/RingSolver.agda`.

Appendix A

Setting up the translate module

Here we will show how to obtain, install, and get started with the translate module. We will assume that Agda has already been set up, and that the user is running Linux or macOS.

First, download the latest version of the source code for the translate module from [16]. If downloaded as an archive, extract it. Move the source code to a suitable location, which we will denote as `TPATH`, where it will be stored permanently. Make sure the contents of `TPATH` include both the `src` directory as well as the `translate.agda-lib` file.

Now, assuming you haven't done so already, create a `.agda` directory in your home directory (and note the dot at the beginning of the directory name). Append `TPATH/translate.agda-lib` to the file `.agda/libraries`, and, if you want the translate module to be available in your Agda projects by default, append `translate` to the file `.agda/defaults`.

The translate module has now been installed, and you can start using it in your projects. A simple template that imports the translate module is as follows:

```
module TranslateTest where

open import Translate
open import Translate.Common
open import Translate.Arithmetic
open import Translate.Combinatorics
open import Translate.EqReasoning
open import Translate.Solver
open import Translate.Tools
open import Translate.Bijection using (getTo; getFrom)

ex : ∀ {a b c} → a * (b + c) ≡ c * a + a * b
ex {a} {b} {c} = begin
  a * (b + c)   ≡⟨ distribˡ-*-+ ⟩
  a * b + a * c ≡⟨ +-comm ⟩
  a * c + a * b ≡⟨ +-cong *-comm refl ⟩
  c * a + a * b ∎
```

```
f : ∀ {a b c} → lift (a * (b + c)) → lift (c * a + a * b)
f = getTo (bijection ex)
```

For further examples, e.g. of how to output bijections as a whole, see `TPATH/examples`. In case you would like to change any aspect of the translate module, you can modify the source code in `TPATH/src`. It is organized as follows:

Translate.agda re-exports the most commonly used functionality.

Translate/Arithmetic.agda contains a small library of simple algebraic identities.

Translate/Base.agda defines equivalence on expressions, the translation method.

Translate/Bijection.agda defines a data type for representing bijections.

Translate/Cancellation.agda implements cancellation procedures.

Translate/Combinatorics.agda contains a small library of combinatorial identities.

Translate/Common.agda gives some common imports.

Translate/EqReasoning.agda defines the equational reasoning environment.

Translate/Properties.agda proves some properties of the equivalence.

Translate/Solver.agda implements the semiring solver.

Translate/Tools.agda gives some handy tools for equivalences and bijections.

Translate/Types.agda defines expressions and helper functions.

School of Computer Science
Reykjavík University
Menntavegur 1
101 Reykjavík, Iceland
Tel. +354 599 6200
Fax +354 599 6201
www.ru.is