Formalizing the translation method in Agda

Bjarki Ágúst Guðmundsson

MSc in Computer Science, thesis defence

School of Computer Science
Reykjavík University

HÁSKÓLINN Í REYKJAVÍK
REYKJAVIK UNIVERSITY

Introduction

Bijections

- Given two sets of combinatorial objects, P and Q, do they have the same cardinality?
- If yes, find a bijection between P and Q
- Clearly possible if and only if $|P| = |Q|$

- Usually we don't just want any such bijection
- When such a bijection is defined in a "natural" way, it will often give insight into *why* the two sets have the same cardinality
- This will help further study on these, as well as related sets of combinatorial objects

Methods

- Many combinatorialists have based their careers on finding “elegant” bijections
- Finding bijections with nice properties has been considered an art form, requiring both creativity and ingenuity
- Many have sought ways to make it easier to come up with such bijections

Bijection machines: Parameterized bijections that work for a general class of problems

Semi-automatic methods: Methods that help derive a bijection for a given problem, but requires some assistance from the combinatorialist

Fully automatic methods: Some have envisioned completely automatic methods for finding bijections

The translation method

- The translation method has proved to be a very helpful tool for coming up with bijections
- Already been many successful applications

- Unfortunately the method is somewhat loosely defined
- Its current implementation, in Maple, has some caveats
- Cumbersome and clumsy to apply the method in its current form

- We propose a formalization of the translation method in the programming language Agda, along with various extensions
- Hopefully making it easier for combinatorialists to apply the translation method to their own problems

The translation method

The translation method

- Introduced by Wood and Zeilberger [3]

- Given two sets of combinatorial objects, P and Q
- Also given an *algebraic* proof that $|P| = |Q|$
- Want a *natural bijection* between P and Q

- The translation method makes use of the algebraic proof that $|P| = |Q|$, "lifting" the proof to a bijection between P and Q
- If the algebraic proof is "natural", the hope is that the resulting bijection will be natural as well

Lifting an expression

Definition

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.

Example

The integer 2^k can be lifted to the set of binary strings of length k.

- In this way lifting will be used to give a combinatorial interpretation of an expression

Example

The integer 2^k can also be lifted to the set of integers $\{1, \dots, 2^k\}$.

Lifting an expression

Expression E	Lifted E	
c	$[c] = \{0, 1, \dots, c-1\}$	
$a \cdot b$	$A \times B$	
$a + b$	$A \sqcup B$	(labeling elements with L or R)
2^n	binary strings of length n: $\mathbf{2}^n$	
$\binom{n}{k}$	binary strings of length n with exactly k 1's: $\binom{[n]}{k}$	

Table 1: Standard interpretations of common expressions. Here a and b can be lifted to A and B, respectively.

Lifting an expression

Example

The expression $2 \cdot 2^{k-1}$ can be lifted to the set $\{0, 1\} \times \mathbf{2}^{k-1}$.

Lifting an identity

Definition

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.

Example

Consider the identity $2^k = 2 \cdot 2^{k-1}$. We have seen that 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:

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

It's clear that bin_k forms a bijection between $\mathbf{2}^k$ and $\{0,1\} \times \mathbf{2}^{k-1}$, so we can say that the identity $2^k = 2 \cdot 2^{k-1}$ can be lifted to the bijection bin_k.

Lifting an identity

- Lifting also gives us a combinatorial interpretation of identities, but now in terms of bijections
- The translation method is just a way to lift the identity $|P| = |Q|$
- Before giving the translation method, we need some building blocks

Building blocks

- Consider the algebraic proof that $|P| = |Q|$. It may rely on facts such as:
    - commutativity of addition, associativity of multiplication, distributivity of multiplication over addition (the natural numbers form a commutative semiring)
    - definitions, such as $2^n = 2 \cdot 2^{n-1}$
    - congruence of addition and multiplication
    - that $=$ forms an equivalence relation
    - "smaller" versions of the identity being proven, by induction
- Each of these facts, interpreted as identities, can be lifted to a corresponding bijection

Building blocks

Example

Commutativity of addition: $a + b = b + a$. If a, b can be lifted to A, B, respectively, we're looking for a bijection $f : A \sqcup B \to B \sqcup A$.

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

Example

Distributivity of multiplication over addition (from the left): $a \cdot (b + c) = a \cdot b + a \cdot c$. If a, b, c can be lifted to A, B, C, respectively, we're looking for a bijection $f : A \times (B \sqcup C) \to (A \times B) \sqcup (A \times C)$.

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

Building blocks

Identity	Lifted identity
$a = a$	$A \cong A$
$\dfrac{a = b}{b = a}$	$\dfrac{A \cong B}{B \cong A}$
$\dfrac{a = b \qquad b = c}{a = c}$	$\dfrac{A \cong B \qquad B \cong C}{A \cong C}$
$a + b = b + a$	$A \sqcup B \cong B \sqcup A$
$a \cdot b = b \cdot a$	$A \times B \cong B \times A$
$(a + b) + c = a + (b + c)$	$(A \sqcup B) \sqcup C \cong A \sqcup (B \sqcup C)$
$(a \cdot b) \cdot c = a \cdot (b \cdot c)$	$(A \times B) \times C \cong A \times (B \times C)$
$a \cdot (b + c) = a \cdot b + a \cdot c$	$A \times (B \sqcup C) \cong (A \times B) \sqcup (A \times C)$

Building blocks

Identity	Lifted identity
$a + 0 = a$	$A \sqcup \emptyset \cong A$
$a \cdot 1 = a$	$A \times [1] \cong A$
$a \cdot 0 = 0$	$A \times \emptyset \cong \emptyset$
$\dfrac{a = c \qquad b = d}{a + b = c + d}$	$\dfrac{A \cong C \qquad B \cong D}{A \sqcup B \cong C \sqcup D}$
$\dfrac{a = c \qquad b = d}{a \cdot b = c \cdot d}$	$\dfrac{A \cong C \qquad B \cong D}{A \times B \cong C \times D}$
$a^n = a \cdot a^{n-1}$	$A^n \cong A \times A^{n-1}$
$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$	$\binom{[n]}{k} \cong \binom{[n-1]}{k-1} \sqcup \binom{[n-1]}{k}$

The translation method

Definition

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 the previous table.
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 the previous table. If the proof step is using a "smaller" identity of the form $|P| = |Q|$ using induction, then that identity can be lifted recursively using the translation method. If neither the previous table nor recursion can be used, the proof step has to be lifted manually.
3. Compose the lifted bijections for a bijection between P and Q.

Applying the translation method

Example

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

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

Applying the translation method

- Notice that $2^n \cdot 2^n$ and 4^n can be lifted to $\mathbf{2}^n \times \mathbf{2}^n$ and $\mathbf{4}^n$, respectively
  - the set of pairs of binary strings of length n
  - 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 inductive proof
- First we need to decompose our proof into atomic steps

Applying the translation method

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

- Next we need to lift each of the atomic steps to a bijection

Applying the translation method

Lifting $2^n \cdot 2^n = (2 \cdot 2^{n-1}) \cdot (2 \cdot 2^{n-1})$

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.

The identity $2^n = 2 \cdot 2^{n-1}$ can be lifted to the bijection bin_n. Congruence of multiplication can be lifted to the (parameterized) bijection

$$\text{mult-cong}_{f,g} \ : A \times B \to C \times D$$

where $f \ : A \to C$ and $g \ : B \to D$ are bijections.

Combining these, we see that the current step can be lifted to the bijection

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

Applying the translation method

Lifting $(2 \cdot 2^{n-1}) \cdot (2 \cdot 2^{n-1}) = ((2 \cdot 2^{n-1}) \cdot 2) \cdot 2^{n-1}$

This is just associativity of multiplication, which we know can be lifted to the bijection mult-assoc.

One has to be careful, 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 $(a \cdot b) \cdot c = a \cdot (b \cdot c)$. Hence

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

Applying the translation method

Lifting $((2 \cdot 2^{n-1}) \cdot 2) \cdot 2^{n-1} = (2 \cdot (2 \cdot 2^{n-1})) \cdot 2^{n-1}$

This is just 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):

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

Applying the translation method

Lifting $(2 \cdot (2 \cdot 2^{n-1})) \cdot 2^{n-1} = ((2 \cdot 2) \cdot 2^{n-1}) \cdot 2^{n-1}$

This is just a nested application of associativity of multiplication, so similar to the last two steps, this step can be lifted to

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

Applying the translation method

Lifting $((2 \cdot 2) \cdot 2^{n-1}) \cdot 2^{n-1} = (2 \cdot 2) \cdot (2^{n-1} \cdot 2^{n-1})$

This is just associativity of multiplication, and this time it is of the same form as $\mathrm{mult\text{-}assoc}$. Hence this step can be lifted to

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

Applying the translation method

Lifting $(2 \cdot 2) \cdot (2^{n-1} \cdot 2^{n-1}) = 4 \cdot (2^{n-1} \cdot 2^{n-1})$

Here we're making use of the fact that $2 \cdot 2 = 4$, on the left side of the multiplication. We have not seen this identity before, so we'll have to lift it manually. We want 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. Which one is "correct"?

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

Applying the translation method

Lifting $4 \cdot (2^{n-1} \cdot 2^{n-1}) = 4 \cdot 4^{n-1}$

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.

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

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

Applying the translation method

Lifting $4 \cdot 4^{n-1} = 4^n$

The symmetry of this identity is $4^n = 4 \cdot 4^{n-1}$, which follows from the definition of exponentiation. For completeness, the lifted bijection, $\mathrm{quad}_k : \mathbf{4}^k \to [4] \times \mathbf{4}^{k-1}$, is as follows:

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

We can lift this step to

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

Applying the translation method

- Now that we've lifted the individual steps to bijections, the last step of the translation method 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:
  - Some of the identities in the intermediate steps assumed $n \geq 1$
  - We are applying the translation method recursively, but currently have no base case
- Both of these issues can be fixed by treating $n = 0$ as a base case. There is a unique bijection $\mathbf{2}^0 \times \mathbf{2}^0 \cong \mathbf{4}^0$:

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

Applying the translation method

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

s	$\Phi(s)$
$(01, 00)$	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

Applying the translation method

s	$\Phi(s)$
$(0000101011, 0010111111)$	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

Applying the translation method

- The bijection produced is not direct, but rather a composition of many simpler bijections
- The resulting bijection is not immediately useful, but is rather intended to be experimentally used by the combinatorialist to discover a direct bijection

- Applying the translation method is very tedious to do by hand
- Luckily the tedious parts are very mechanical, and can be easily performed by a computer
- Wood and Zeilberger noticed this, and gave an implementation in Maple

Implementation

- There are some issues with their implementation, however:
    - The conversion from the algebraic proof to code is unintuitive
    - Maple is a dynamically-typed language, making it easy to make small mistakes
    - Maple is proprietary software, and one has to buy a license to use it
    - Addition gives an error if the given sets are not disjoint $(2 \cdot a = a + a)$
    - When specifying bijections explicitly, as we did for the base cases, nothing to check that they are indeed bijections
    - Taking this to the extreme, there's no guarantee that their implementation behaves as advertised

Implementation

- We started implementing the translation method in a statically-typed language to address these issues
- Soon realized that static typing was not enough to address all the issues:
  - binary strings vs. integer compositions
  - binary strings of length 5 vs. binary strings of length 10
- This suggested using a programming language that supports *dependent types*
  - Types that can depend on values
  - binary strings of length 5 vs. binary strings of length 10
  - Much more powerful than that: allows one to do formal proofs

Implementation

- We looked at two general-purpose programming languages that supported dependent types
  - Agda
  - Idris
- Goal: Implement the translation method in Agda, addressing the issues that we found in the previous implementation, possibly extending the method with some helpful features

Agda

Agda

- Agda is a pure functional programming language, similar in many respects to Haskell, that supports dependent types
- Because of its support for dependent types, it's possible to state and prove formal theorems as Agda code
- It also has some simple tools to help with that, and can therefore be considered a proof assistant

Algebraic proofs in Agda

- Agda's standard library comes with a handful of theorems about algebraic properties

```
+-comm : ∀ a b → a + b ≡ b + a

*-assoc : ∀ a b c → (a * b) * c ≡ a * (b * c)

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

- These can then be combined to prove more complex identities
- Consider $(b + c) \cdot a = c \cdot a + a \cdot b$

Proofs in Agda

```
ex : ∀ {a b c} → (b + c) * a ≡ c * a + a * b
ex {a} {b} {c} = begin
  (b + c) * a   ≡⟨ ? ⟩
  c * a + a * b ∎
```

Proofs in Agda

```
ex : ∀ {a b c} → (b + c) * a ≡ c * a + a * b
ex {a} {b} {c} = begin
  (b + c) * a   ≡⟨ distribʳ-*-+ a b c ⟩
  b * a + c * a ≡⟨ ? ⟩
  c * a + a * b ∎
```

Proofs in Agda

```
ex : ∀ {a b c} → (b + c) * a ≡ c * a + a * b
ex {a} {b} {c} = begin
  (b + c) * a   ≡⟨ distribʳ-*-+ a b c ⟩
  b * a + c * a ≡⟨ +-comm (b * a) (c * a) ⟩
  c * a + b * a ≡⟨ ? ⟩
  c * a + a * b ∎
```

Proofs in Agda

```
ex : ∀ {a b c} → (b + c) * a ≡ c * a + a * b
ex {a} {b} {c} = begin
  (b + c) * a   ≡⟨ distribʳ-*-+ a b c ⟩
  b * a + c * a ≡⟨ +-comm (b * a) (c * a) ⟩
  c * a + b * a ≡⟨ cong (λ x → c * a + x) (*-comm b a) ⟩
  c * a + a * b ∎
```

The translate module

The translate module

- Idea: Ability to take an existing algebraic proof in Agda, and apply the translation method directly to it (or by changing as little as possible)
- This is not immediately possible: given something of type `a ≡ b`, there is no way to access the individual proof steps
- Reimplemented the basic data types in Agda, storing the relevant information where necessary

The translate module

- Expressions are now stored in an abstract syntax tree, `Expr`
  - Calling the `lift` function gives the lifted to a set
- The equivalence relation, ≡, maintains bijection-related information
  - Calling the `bijection` function performs the translation method, returning the lifted bijection
- We then proved, again, many of the existing algebraic properties under our new definition of expressions and equivalence, augmented with their associated lifted bijections
- This further allowed us to prove that our expressions formed a commutative semiring under our equivalence relation

Using the translate module

- Just as with Agda, we can now use these simple algebraic properties to prove more complex identities
- Say we want to prove $2^n \cdot 2^n = 4^n$
- Towards that, let's begin by simply proving $2 \cdot 2 = 4$

```
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))) ∷ []
  )))
```

Using the translate module

- Now we can go on to prove our identity, starting with the base case:

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

- For the inductive case, recall our previous inductive proof

Using the translate module

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

Using the translate module

```
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)                     ∎
```

Using the translate module

- After proving this, it now becomes as simple as calling `bijection phi` to get the translated bijection!
- This will give us the same bijection as when we applied the translation method by hand

Semiring solver

Tools for semirings

- As we saw, our expressions form a commutative semiring under our new definition of equivalence
- While that is interesting in its own right, we wondered if we could use that fact in interesting ways
- Tools for automatically proving equalities in commutative semirings, often known as semiring solvers

Semiring solver

- From a bird's-eye view, a semiring solver works as follows:

$$\begin{tikzcd}
f \arrow[d, "\texttt{normalize}"'] & g \arrow[d, "\texttt{normalize}"] \\
f' \arrow[r, leftrightarrow, "\mathsf{refl}"'] & g'
\end{tikzcd}$$

- Given two expressions, the solver brings them both to normal form, and then asks the user to prove that the normal forms are equivalent
- If the two expressions are equivalent, their normal forms will often be identical, in which case a simple `refl` will do

Semiring solver

- The state of art in semiring solvers seems to be given by Grégoire and Mahboubi [2]
- This is currently the semiring solver used by Coq
- It has been partially ported to Agda, but it turns out that this implementation is not general enough to support our semiring
- Instead of trying to interface with Coq, or fixing Agda's implementation, we decided to make our own implementation specifically for our semiring loosely based on [2]
    - Simpler, but less efficient normal form

Semiring solver

- We could nevertheless make use of Agda's solver interface, so we only had to implement the normalize function, and prove its correctness:

```
normalize : ∀ {n}
          → :Expr n
          → RightLeaningSumOfNormalizedMonomials n

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

Semiring solver

- Consider the identity $(a+b)^2 = a^2 + 2ab + b^2$
- Usually straightforward to prove, but tedious
- Using the solver:

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

Semiring solver

- If we have two equivalent expressions that only contain variables, constants, addition and multiplication, the solver will always give identical normal forms
- If it includes other constructs, such as functions, it may or may not give identical normal forms

Cancellation

Cancellation

- So far we have ignored operations such as subtraction and division
  - Working with natural numbers in Agda
  - These numbers are supposed to represent set cardinalities
- If we have the expression $a - b$, what is $A \setminus B$ supposed to represent if $B \nsubseteq A$?
- Wood and Zeilberger came up with a way to make sense of this in the translation method
- In an earlier paper, Feldman and Propp [1] developed a theory of "cancellation procedures"
  - Turns out to give the same approach for subtraction as Wood and Zeilberger's
  - Also supports other operations

Cancellation

- We have a function f that somehow adds structure to the input set
  - $f_1(S) = S \sqcup C$
  - $f_2(S) = S \times C$
  - $f_3(S) = S^k$
  - $f_4(S) = 2^S$
- Given a bijection $f(A) \cong f(B)$, recover a bijection $A \cong B$, essentially "cancelling" the f construction
  - Given a bijection $A \sqcup C \cong B \sqcup C$, recover a bijection $A \cong B$
- Feldman and Propp's theory tells us when a function is cancellable
  - Possible for f_1 and f_3 without any conditions, f_2 if C has a distinguished element, but f_4 is not cancellable
  - They also provide polynomial-time "cancellation procedures" for f_1, f_2 and f_3

Cancellation of addition

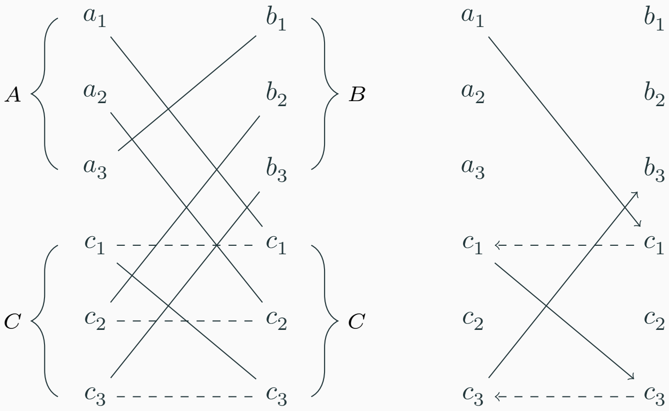
- The cancellation procedure for addition is as follows:

Definition (Cancellation of addition)

Given an $a \in A$, as well as a bijection $f : A \sqcup C \to B \sqcup C$, produce an element of B as follows:

1. If $f(a) \in B$, return $f(a)$. Otherwise let $c = f(a)$ and continue:
2. If $f(c) \in B$, return $f(c)$. Otherwise let $c = f(c)$, and go back to step 2.

Cancellation of addition



- The resulting bijection $A \cong B$ will have $a_1 \mapsto b_3$, $a_2 \mapsto b_2$ and $a_3 \mapsto b_1$

Cancellation of addition

- We implemented cancellation of addition in Agda:

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

- Trivial to implement in many programming languages, but not in Agda:
    - We have to prove that this forms a bijection
    - Agda needs to recognize that the program terminates, to make sure logic is sound
        - Halting problem is undecidable
        - Heuristics used to recognize if a program terminates
- Yet to implement other cancellation procedures, but believe the same techniques can be used

Conclusions and future work

Conclusions

- We did the following:
    - Gave a formal implementation of the translation method in Agda
    - Implemented a semiring solver that can be used in conjunction with the translation method
    - Implemented the cancellation procedure for addition in our translate module
    - Showed how division and k-th roots can be handled when applying the translation method
- We believe the translate module is ready for use by other combinatorialists, and we hope this will make it easier to apply the translation method to their own problems

Future work

- Despite that, there are still things that can be improved
  - Would be nice to be able to write `nat 3` simply as `3`
  - There is some code duplication due to the solver
  - Current design makes it a bit difficult to add new functions and data types
  - Implement the remaining cancellation procedures

Thanks!

References I

David Feldman and James Propp.
Producing new bijections from old.
Advances in Mathematics, 113(1):1–44, 1995.

Benjamin Grégoire and Assia Mahboubi.
Proving equalities in a commutative ring done right in Coq.
In *International Conference on Theorem Proving in Higher Order Logics*, pages 98–113. Springer, 2005.

Philip Matchett Wood and Doron Zeilberger.
A translation method for finding combinatorial bijections.
Annals of Combinatorics, 13(3):383, 2009.