CS 173 Lecture 13: Bijections and Data Types

José Meseguer

University of Illinois at Urbana-Champaign

1 More on Bijections

Bijecive Functions and Cardinality. If A and B are finite sets we know that $f : A \to B$ is *bijective* iff $|A| = |B|$.

If A and B are *infinite* sets we can *define* that they have the *same cardinality*, written $|A| = |B|$ iff there is a *bijective* function. $f : A \to B$.

This agrees with our intuition since, as f is in particular *surjective*, we can use $f : A \to B$ to "list[1]" *all* elements of B by elements of A. The reason why writing $|A| = |B|$ makes sense is that, since $f : A \to B$ is also bijective, we can also use $f^{-1} : B \to A$ to "list" *all* elements of A by elements of B. Therefore, this captures the notion of A and B having the "same degree of infinity," since their elements can be put into a *bijective* correspondence (also called a *one-to-one and onto* correspondence) with each other.

We will also use the notation $A \cong B$ as a shorthand for the existence of a bijective function $f : A \to B$. Of course, then $A \cong B$ iff $|A| = |B|$, but the two notations emphasize sligtly different, though equivalent, intuitions. Namely, $A \cong B$ emphasizes the idea that A and B can be placed in *bijective correspondence*, whereas $|A| = |B|$ emphasizes the idea that A and B have the *same cardinality*. In summary, the notations $|A| = |B|$ and $A \cong B$ mean, by definition:

$$A \cong B \ \Leftrightarrow_{def} \ |A| = |B| \ \Leftrightarrow_{def} \ \exists f \in [A{\to}B](f \ \textit{bijective})$$

Arrow Notation and Arrow Composition. We will adopt the following *arrow notation* to abbreviate the description of injective, surjective, and bijective functions:

- $f : A \rightarrowtail B \ \Leftrightarrow_{def} \ f : A \to B$ and f *injective*
- $f : A \twoheadrightarrow B \ \Leftrightarrow_{def} \ f : A \to B$ and f *surjective*
- $f : A \xrightarrow{\cong} B \ \Leftrightarrow_{def} \ f : A \to B$ and f *bijective.*

Theorem 1 (Arrow Composition).

1. Given $A \overset{f}{\rightarrowtail} B \overset{g}{\rightarrowtail} C$ then $g \circ f : A \rightarrowtail C$

[1] When $A = \mathbb{N}$, such a listing is called an *enumeration*. But for sets with greater degree of infinity, like when $A = \mathbb{R}$, the word "listing" is more general and avoids restricting the notion to countably infinite sets.

2. Given $A \overset{f}{\twoheadrightarrow} B \overset{g}{\twoheadrightarrow} C$ then $g \circ f : A \twoheadrightarrow C$
3. Given $A \overset{f}{\overset{\simeq}{\rightarrow}} B \overset{g}{\overset{\simeq}{\rightarrow}} C$ then $g \circ f : A \overset{\simeq}{\rightarrow} C$

Furthermore, for any set A, $id_A : A \rightarrow A$ is bijective.

Proof: (2) was proved in Feb. 23 Discussion Session. To see (1), note that, since f and g are injective,

$$g(f(x)) = g(f(x')) \Rightarrow f(x) = f(x') \Rightarrow x = x'.$$

(3) follows immediately from (1) and (2).

$id_A : A \rightarrow A$ is bijective because $id_A = id_A^{-1}$. This finishes the proof of the theorem.

The Set of Bijective Functions $[A \rightarrow B]_{\cong}$. The set of bijective functions from A to B is denoted $[A \rightarrow B]_{\cong}$ and is defined by:

$$[A \rightarrow B]_{\cong} =_{def} \{f \in [A \rightarrow B] \mid f \ \textit{bijective}\}$$

Of course, $[A \rightarrow B]_{\cong} \neq \varnothing$ iff $|A| = |B|$.

Given any set A, a bijection $f : A \overset{\simeq}{\rightarrow} A$ is called a a *permutation*. Therefore, the *set of permutations* of A is defined as:

$$Perm(A) =_{def} [A \rightarrow A]_{\cong}$$

The slides for this lecture contain a simple example of a set of permutations for a finite set A with $|A| = 3$ and show that $|Perm(A)| = 6 = 3!$ for such a set. This is an instance of the following theorem:

Theorem 2. Given finite sets with $|A| = |B| = n$, then $|[A \rightarrow B]_{\cong}| = n!$ In particular, $|Perm(A)| = n!$

The proof is by induction on $n = |A| = |B|$ and is left as an exercise.

Algebraic Properties of $[A \rightarrow A]$ and $Perm(A)$. Since $Perm(A) \subseteq [A \rightarrow A]$, we should consider the algebraic properties of $Perm(A)$ within the contex of those of $[A \rightarrow A]$.

The obvious *operation* that we can perform in $[A \rightarrow A]$ is *function composition.* Given $f, g \in [A \rightarrow A]$ we *know* from Theorem 3 in Lecture 11 that $f \circ g \in [A \rightarrow A]$. What *algebraic properties* does this operation have? We also *know* from the same theorem that for $f, g, h \in [A \rightarrow A]$ we have:

Associativity. $(f \circ g) \circ h = f \circ (g \circ h)$
Identity. $id_A \circ f = f = f \circ id_A$.

These are called the *monoid laws*. Therefore, $[A \rightarrow A]$ is a *monoid* for the function composition operation.

What are the algebraic properties of $Perm(A)$? By the above Arrow Composition Theorem we *know* that given $f, g \in Perm(A)$, $f \circ g \in Perm(A)$. And by Theorem 3 in Lecture 11 and Lemma 1 in Lecture 12 we also *know* that for $f, g, h \in Perm(A)$ the following algebraic properties hold:

Associativity. $(f \circ g) \circ h = f \circ (g \circ h)$
Identity. $id_A \circ f = f = f \circ id_A$.
Inverse. $f \circ f^{-1} = id_A = f^{-1} \circ f$.

These are called the *group laws*. Therefore, $Perm(A)$ is a *group* for the permutation composition operation.

2 Bijections as Changes of Data Representation

Algorithms and programs manipulate *data*. But data can be represented in different ways. For example, we can represent:

- *true* as **T** or as 1
- *false* as **F** or as 0

We can use *bijections* to change data representations. For example, the bijection $letter2number : \{\mathbf{T}, \mathbf{F}\} \xrightarrow{\cong} \{0, 1\}$ defined by: $letter2number =_{def} \lambda x \in \{\mathbf{T}, \mathbf{F}\}.\ \mathbf{if}\ x = \mathbf{T}\ \mathbf{then}\ 1\ \mathbf{else}\ 0\ \mathbf{fi} \in \{0, 1\}$ allows us to change the truth values from letters to numbers; and its inverse, $number2letter =_{def} (letter2number)^{-1}$ allows us to change it back.

Likewise, the natural numbers do have different data representations. For example, we saw in the slides for Lecture 9 that the natural numbers in *decimal notation* are a language

$$dec\mathbb{N} \subset \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}^*$$

specifiable by a grammar.

In an entirely similar way, the natural numbers in *binary notation* are also a language

$$bin\mathbb{N} \subset \{0, 1\}^*$$

specifiable by a grammar.

An even simpler notation for numbers is the *finger notation*, where $finger\mathbb{N}$ is, by definition, the string set $finger\mathbb{N} =_{def} \{|\}^*$, so that we repesent numbers by "fingers" as follows:

$$0 = \epsilon,\ 1 = |,\ 2 = ||,\ 3 = |||, \ldots$$

and number addition is "finger concatenation"

$$3 + 2 = (|||)\,(||) = |||||$$

We can of course *convert* numbers in one represetation into the same numbers in a *different* representation by the well-known decimal to binary conversion and back, and likewise we can convert them into finger notation. That is, we have, for example, bijections:

- $dec2bin : dec\mathbb{N} \xrightarrow{\simeq} bin\mathbb{N}$, with inverse $bin2dec : bin\mathbb{N} \xrightarrow{\simeq} dec\mathbb{N}$
- $bin2finger : bin\mathbb{N} \xrightarrow{\simeq} finger\mathbb{N}$, with inverse $finger2bin : finger\mathbb{N} \xrightarrow{\simeq} bin\mathbb{N}$

Furthermore, we can *compose* these changes of data representation to ger new ones. For example, composing $dec2bin : dec\mathbb{N} \xrightarrow{\simeq} bin\mathbb{N}$ with $bin2finger : bin\mathbb{N} \xrightarrow{\simeq} finger\mathbb{N}$ we get a data conversion

$$dec2finger = bin2finger \circ dec2bin : dec\mathbb{N} \xrightarrow{\simeq} finger\mathbb{N}.$$

The subsets of a set are also data manipulated by many algorithms. Given a finite set A, its subsets $X \in \mathcal{P}(A)$ are precisely such data elements. But subsets of $A = \{a_1, \ldots, a_n\}$ can be represented in differen ways. The obvious one is to represent $B \in \mathcal{P}(A)$ by itself, i.e., as the *set* of its k elements $B = \{a_{i_1}, \ldots, a_{i_k}\}$. But an attractive, alternative representation is to represent B as a *predicate*, that is, as a truth-valued function. Specifically, we can represent each $B \in \mathcal{P}(A)$ by its so-called *characteristic function*, which is the following predicate:

$$\chi_B =_{def} \lambda x \in A.\ (x \in B) \in \{\mathbf{T}, \mathbf{F}\}.$$

The change of data representation

$$B \mapsto \chi_B$$

is a bijection

$$subset2pred : \mathcal{P}(A) \xrightarrow{\simeq} [A \to \{\mathbf{T}, \mathbf{F}\}]$$

where $subset2pred =_{def} \lambda B \in \mathcal{P}(A).\ \chi_B \in [A \to \{\mathbf{T}, \mathbf{F}\}]$. It inverse is the function

$$pred2subset =_{def} \lambda p \in [A \to \{\mathbf{T}, \mathbf{F}\}].\ \{x \in A \mid p(x) = \mathbf{T}\} \in \mathcal{P}(A).$$

Checking that, indeed, $pred2subset = (subset2pred)^{-1}$ is left as a useful exercise.

3 What is a Data Type?

The changes of data representation we have considered are changes in the representation of *data types* as used in programming languages. For example, $letter2number : \{\mathbf{T}, \mathbf{F}\} \xrightarrow{\simeq} \{0, 1\}$ is a change of representation for the data type of *Booleans*, $dec2bin : dec\mathbb{N} \xrightarrow{\simeq} bin\mathbb{N}$ is a change of representation for the data type of *Naturals*, and $subset2pred : \mathcal{P}(A) \xrightarrow{\simeq} [A \to \{\mathbf{T}, \mathbf{F}\}]$ is a change of representation for the data type of *FiniteSubsets* of A.

Data types are of two kinds:

Built-in data types like *Booleans*, *Naturals*, *Integers*, and *Floats* are usally provided by a programming language in a built-in way;

User-definable data types like *Lists*, *FiniteSubsets* of a set, *BinaryTrees*, *FiniteFunctions* (called *map* data types), *FiniteRelations* (called *tables* in databases), *FiniteGraphs*, and so on, are programmed by the user, or belong to libraries such as the C++ template library.

The *elements* of a data type are called *data elements* or *data structures*.

All *algorithms* manipulate data structures in given data types. In fact they are *classified* by the kind of data types they handle. For example, as:

- *numerical* algorithms,
- *string* algorithms,
- *tree* algorithms,
- *graph* algorithms,

and so on.

It is of course *impossible* to mathematically verify (as opposed to just testing) the *correctness* of algorithms and programs without having *mathematical models* of the data types they manipulate. Therefore the question:

> *What is a data type?*

which could be rephrased as:

> *How should a data type be modeled?*

is not an idle or trivial question at all. Without a satisfactory, mathematical answer to this question it is impossible to reason mathematically about data types and the correctness of programs and algorithms.

Question1: What is a Data Type? Since a data type is a collection of data elements and Set Theory is the mathematical theory of collections of objects, a possible, tentative answer to this questions could be:

> **Answer1:** *A data type is just a set of data.*

This sounds quite reasonable. For example, a so-called *enumeration type* consisting of data elements a, b, c, and d can be modeled mathematically as the *set* $\{a, b, c, d\}$.

But is this answer right? And how can we *find out* whether it is right or not? One way to find out is as follows. Whatever answer we give to **Question 1**, such an answer should be *consistent* with an answer to the following, closely related question:

> **Question 2:** *What is a change of data representation between equivalent data types?*

where by "equivalent" we mean that, for example, $\{\mathbf{T}, \mathbf{F}\}$ and $\{0, 1\}$ are equivalent representations of the *Booleans*, and that $dec\mathbb{N}$ and $bin\mathbb{N}$ are equivalent representations of the *Naturals*. For all the examples we have seen in Section 2, the most obvious, tentative answer is:

> **Answer2:** *A change of data representation between equivalent data types D and D' is just a bijection $f : D \xrightarrow{\cong} D'$.*

This also seems reasonable, since all the changes of data representation we have considered *are* bijections.

But is this answer right? And how can we *find out* whether it is right or not? One way to find out is to *test* whether this answer is right in the *simplest possible* example, namely, the equivalent data types $\{\mathbf{T}, \mathbf{F}\}$ and $\{0, 1\}$, where, of course, in $\{0, 1\}$ *not*, *or*, and the *and* operations have the following truth tables:

NOT :

X	$\neg X$
1	0
0	1

OR :

X_1	X_2	$X_1 \vee X_2$
1	1	1
1	0	1
0	1	1
0	0	0

AND :

X_1	X_2	$X_1 \wedge X_2$
1	1	1
1	0	0
0	1	0
0	0	0

Therefore, if **Answer2** is right, the following bijection

$$\mathit{flip} : \{\mathbf{T}, \mathbf{F}\} \xrightarrow{\cong} \{0, 1\}$$

where $\mathit{flip} =_{def} \lambda x \in \{\mathbf{T}, \mathbf{F}\}.\ \mathbf{if}\ x = \mathbf{T}\ \mathbf{then}\ 0\ \mathbf{else}\ 1\ \mathbf{fi} \in \{0, 1\}$ should be a change of data representation. But is this right? It does not seem so, since we get the following *nonsense!*

1. $\mathbf{T} \wedge \mathbf{T} = \mathbf{T}$ and $\mathit{flip}(\mathbf{T}) \wedge \mathit{flip}(\mathbf{T}) = 0 \wedge 0 = 0$,
2. $\mathbf{T} \wedge \mathbf{F} = \mathbf{F}$ and $\mathit{flip}(\mathbf{T}) \wedge \mathit{flip}(\mathbf{F}) = 0 \wedge 1 = 0$.

Therefore, our tentative answers:

> **Answer1:** *A data type is just a set of data.*
>
> **Answer2:** *A change of data representation between equivalent data types D and D' is just a bijection $f : D \xrightarrow{\cong} D'$.*

are completely wrong!

The Problem. What is the problem? aren't data types sets and changes of data representation bijections? Yes, they are. But the above example painfully shows that they *cannot* be *just* sets and *just* bijections between them. We must look for *shaper* answers of the form:

> **Answer1:** *A data type is a set of data plus??*

> **Answer2:** *A change of data representation between equivalent data types D and D' is a bijection $f : D \xrightarrow{\cong} D'$ plus??*

Let us begin by trying to find what additional requirements we should impose on a bijection to get a satisfactory answer to **Question2**. The place to look at is the nonsense (1) and (2) above. The problem with (2), for example, is that

$$flip(\mathbf{T} \wedge \mathbf{F}) = flip(\mathbf{F}) = 1 \neq 0 = flip(\mathbf{T}) \wedge flip(\mathbf{F})$$

That is, the problem is that the *Boolean operations* are *not* preserved! Any change of Boolean data representation $f : \{\mathbf{T}, \mathbf{F}\} \xrightarrow{\cong} \{0, 1\}$ worth its salt *must* satisfy at least the following requirements:

- $f(\neg x) = \neg(f(x))$
- $f(x \vee y) = f(x) \vee f(y)$
- $f(x \wedge y) = f(x) \wedge f(y)$

Have we seen something like this before? Yes, we have. Changing $\vee$ to $+$ and $\wedge$ to $\_ \cdot \_$, we saw in the **Homomorphism Lemma** of Lecture 7 that the function:

$$\rho : \mathbb{Z} \to \mathbb{Z}_n$$

where $\rho =_{def} \lambda x \in \mathbb{Z}.\ rem(x, n) \in \mathbb{Z}_n$ satisfies:

1. $\rho(a + b) = \rho(a) +_n \rho(b)$
2. $\rho(ab) = \rho(a) \cdot_n \rho(b)$

and that a function preserving operations, such as in this case $\_ + \_$ and $\_ \cdot \_$, is called a *homomorphism.* This suggests the following answer to our two questions:

> **Answer1:** *A data type is a set D plus some operations on that data.*

> **Answer2:** *A change of data representation between equivalent data types D and D' is a bijection $f : D \xrightarrow{\cong} D'$ that is also a homomorphism for the data operations.*

However, **Answer1** is not tight enough. A data type cannot be just any set D with some operations on it. It must be a set whose elements are *representable by finite data structures on a computer*. Furthermore the data operations should be *computable* by means of terminating programs. Technically this is called a *computable* set with *computable* operations. This excludes sets like $\mathbb{R}$ because its data structures are *infinite*. Real numbers can only be *approximated* up to some level of precision in a computer; for example, by using the data type of IEEE floating point numbers. A fortiori, even bigger sets like $\mathcal{P}(\mathbb{R})$ are excluded from **Answer1**: they are not computable at all. **Answer2** should also be tightened: f should be a *computable* function.