Discrete Maths

Philippa Gardner

These lecture notes are based on previous notes by Iain Phillips.

This short course introduces some basic concepts in discrete mathematics: sets, relations, and functions. These notes are written to accompany the slides: the slides summarise the content of the course; the notes provide more explanation and detail. I hope you will find them helpful. You will find that a knowledge of the concepts covered here will be important in understanding many areas of computer science, such as data types, databases, specification, functional programming and logic programming.

Recommended books

1. K.H. Rosen. Discrete Mathematics and its Applications, McGraw Hill 1995.

2. J.L. Gersting. Mathematical Structures for Computer Science, Freeman 1993.

3. J.K. Truss. Discrete Mathematics for Computer Science, Addison-Wesley 1991.

4. R. Johnsonbaugh, Discrete Mathematics, 5th ed. Prentice Hall 2000.

5. C. Schumacher, Fundamental Notions of Abstract Mathematics, Addison-Wesley, 2001.

Related courses include the mathematical reasoning courses (logic, program reasoning and discrete maths 2), Haskell and Databases 1. In particular, we will use some of the notation introduced in the logic course:

$$A \wedge B \quad A \vee B \quad \neg A \quad A \rightarrow B \quad A \leftrightarrow B \quad \forall x.A \quad \exists x.\ A.$$

You will be given assessed exercises in week 7 (submission date Tuesday, November 23nd) and week 9 (submission date Tuesday, December 7th). There will be a test at the end of the Christmas term, and an exam at the end of the year. These notes are self-contained, though rather concise. I shall be grateful if any inaccuracies are brought to my notice, plus any suggestions for improving the course material.

1 Motivation

This course investigates some key mathematical concepts: *sets*, *functions* and *relations*. You will probably have come across these concepts before. This course gives a rigorous account, that forms the underpinnings of many courses in the Computer Science degree.

Sets are like types in Haskell. For example, consider the type declaration

```
data Bool = False | True
```

This command declares a type `Bool` with two elements `True` and `False`. This course introduces the abstract concept of set. We talk about the set of Boolean values True and False, the set of natural numbers, the set of real numbers, the set of prime numbers, the set of students taking the Imperial Computer Science course, ... We describe what it means for two sets to be equal, how to construct new sets from old, and analyse properties of these set constructors.

We also describe the notion of a mathematical function, which maps elements of one set to another. Haskell functions can be viewed as mathematical functions, although they also have the additional property that they are *computable*. [In fact, the idea of a computable function can be expressed precisely. For example, Turing machines describe the computable functions, and this theoretical concept became one route to the invention of computers.] You will be hearing more about computable functions during your time at Imperial. Here we lay the ground work by introducing you to mathematical functions.

We will also discuss relations. A natural example is an equality relation between *equivalent* Haskell functions which intuitively behave in the same way. To illustrate what this means, consider the following two Haskell functions:

```
myand :: Bool -> Bool -> Bool
myand False False = False
myand False True = False
myand True False = False
myand True True = True
```

```
myand' :: Bool -> Bool -> Bool
myand' False x = False
myand' True x = x
```

If the only expressions of type `Bool` are `True` and `False`, then `myand` and `myand'` behave in the same way in the sense that they give the same results on all inputs. We can however declare expressions of type `Bool` which never terminate. For example, consider the declaration

```
bottom :: Bool
bottom = bottom
```

The functions `myand` and `myand'` do not behave in the same way using `bottom`. The expression (`myand' False bottom`) evaluates to `False`, whereas the expression (`myand False bottom`) does not terminate. Now consider in addition the Haskell function

```
myand'' :: Bool -> Bool -> Bool
myand'' b1 b2 = if b1 then b2 else False
```

The function `myand''` has identical behaviour to `myand'`.

We can describe the relationship between equivalent Haskell functions as follows:

> Two Haskell functions `f` : `A` $\rightarrow$ `B` and `g` : `A` $\rightarrow$ `B` behave in the same way if and only if, for all terms `a` in type `A`, then if `f a` terminates then `g a` terminates and `f a` $=$ `g a`, and if `f a` does not terminate then `g a` does not terminate.

You will be learning more about this story in the operational semantics course in the second year. Here, we explore the abstract concept of relations. We will describe *equivalence relations*, special relations which for example behave like the standard equality on the natural numbers and include the above relationship between equivalent Haskell functions, and *orderings* which for example behave like the 'less than' ordering between natural numbers. We will also describe natural ways of constructing new relations from old, including those used in relational databases.

2 Sets

Our starting point will be the idea of a set, a concept that we shall not formally define. Instead, we shall simply use the intuitive idea that a set is a collection of objects.

DEFINITION 2.1 (INFORMAL)
A *set* is a collection of objects (or individuals) taken from a pool of objects. The objects in a set are also called the *elements*, or *members*, of the set. A set is said to *contain* its elements.

We write $x \in A$ when object x is a member of set A. When x is not a member of A, we write $x \notin A$ or sometimes $\neg(x \in A)$ using notation from the logic course.

Sets can be finite or infinite. In fact, we shall see that there are many different infinite sizes! Finite sets can be defined by listing their elements. Infinite sets can be defined using a pattern or restriction. Here are some examples:

1. the two-element set $\mathsf{Bool} = \{\mathsf{True}, \mathsf{False}\}$, which is analogous to the Haskell data type

   ```
   data Bool = True | False
   ```

2. the set of vowels $V = \{a, e, i, o, u\}$, which is read 'V is the set containing the objects (in this case letters) a, e, i, o, u';

3. an arbitrary (nonsense) set $\{1, 2, e, f, 5,$ Imperial$\}$, which is a set containing numbers, letters and a string of letters with no obvious relationship to each other;

4. the set of natural numbers $\mathcal{N} = \{0, 1, 2, 3, \ldots\}$, which is read '$\mathcal{N}$ is the set containing the natural numbers $0, 1, 2, 3, ...$';

5. the set of integers $\mathcal{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$;

6. the set of primes $P = \{x \in \mathcal{N} : x$ is a prime number$\}$;

7. the empty set $\emptyset = \{\ \}$, which contains no elements;

8. nested sets, such as the set $\{\{\emptyset\}, \{a, e, i, o, u\}\}$ containing the sets $\{\emptyset\}$ and $\{a, e, i, o, u\}$ as its two elements.

The set $\mathcal{N}$ is of course an infinite set. The '...' indicates that the remaining elements are given by some rule, which should be apparent from the initial examples: in this case, the rule is to add one to the previous number. Notice that sets can themselves be members of other sets, as the last example illustrates. There is an important distinction between $\{a, b, c\}$ and $\{\{a, b, c\}\}$ for example, or between $\emptyset$ and $\{\emptyset\}$.

2.1 Comparing Sets

We define the notion of one set being a subset of (contained in) another set, and the related notion of two sets being equal. You will be familiar with the notion of sublist from the Haskell course:

```
h:: [Int] -> Int -> [Int]
 h xs n = filter (<n) xs
```

The function `h` takes a list of integers and an integer `n`, and returns a sublist of elements less than `n`.

2.1.1 Subsets

DEFINITION 2.2 (SUBSETS)
Let A, B be any two sets, Then A is a *subset* of B, written $A \subseteq B$, if and only if all the elements of A are also elements of B: that is,

$$A \subseteq B \Leftrightarrow \forall \text{ objects } x.(x \in A \rightarrow x \in B)$$

We have written the subset definition in two ways, using English and also using logical notation which will be familiar from the lectures on logic. Either style is appropriate for this course. Just use whichever suits you best. Notice that the definition talks about all objects x, although we have not said what an object is! We assume that there is an underlying universe of discourse when discussing sets: that is, the set of all possible objects under discussion. This set is sometimes written U.

Any set is a subset of itself. Other simple examples are

$$\begin{aligned} \{a, b\} &\subseteq \{a, b, c\} \\ \{c, c, b\} &\subseteq \{a, b, c, d\} \\ \mathcal{N} &\subseteq \mathcal{Z} \\ \emptyset &\subseteq \{1, 2, 5\} \end{aligned}$$

This last example is tricky. To convince ourselves that it is true we need to show that every element in $\emptyset$ is contained in $\{1, 2, 5\}$. Since there are *no* elements in $\emptyset$, this property is vacuously true: $\emptyset$ is a subset of *every* set!

PROPOSITION 2.3
Let A, B, C be arbitrary sets. If $A \subseteq B$ and $B \subseteq C$ then $A \subseteq C$.

Notice that we have stated that this property holds for arbitrary (all) sets A, B and C: such properties are *universal* properties on sets, in the sense that they are hold for all sets. What does it mean to convince ourselves that such properties are true? In the logic course, you have been exploring very formal definitions of a proof within a logical system. In this course, we do not have to be quite so formal. However, the proofs should be convincing even though they are not written within a purely logical setting. When faced with constructing a proof, check three things: (1) that the arguments put forward are all true and the sequence follows logically from beginning

to end; (2) that the arguments are sufficient to prove the theorem; and (3) that the arguments are all necessary to prove the statement.

Proof of proposition 2.3 Assume that A, B and C are arbitrary sets and that $A \subseteq B$ and $B \subseteq C$. We need to show that, for an arbitrary element x, if x is a member of A then it must also be a member of C. Assume $x \in A$. By assumption, we know that $A \subseteq B$, and hence by the definition of the subset relation that $x \in B$. We also know that $B \subseteq C$, and hence $x \in C$ as required. □

2.1.2 Set Equality

Whenever we introduce a structure (in this case sets), an important part of knowing what we mean is to be able to identify when two such structures are equal, in other words when they denote the same thing. Two sets are equal if and only if they contain the same elements.

DEFINITION 2.4 (SET EQUALITY)
Let A, B be any two sets. Then A *equals* B, written $A = B$, if and only if $A \subseteq B$ and $B \subseteq A$: that is,

$$A = B \Leftrightarrow A \subseteq B \wedge B \subseteq A$$

This definition of equality on sets means that the number of occurrences of elements and the order of the elements of a set do not matter: the sets $\{a, b, c\}$ and $\{b, a, a, c\}$ are equal. Contrast this property with the (perhaps more familiar) list data structure, where the order and number of elements is important.

2.2 Constructing Sets

There are several ways of describing sets. So far, we have informally introduced two approaches: (1) either we just list the elements inside curly brackets:

$$V = \{a, e, i, o, u\}, \quad \mathcal{N} = \{0, 1, 2, \ldots\}, \quad \{\emptyset, \{a\}, \{b\}, \{a, b\}\};$$

or (2) we define a set by stating the property that its elements must satisfy:

$$\begin{aligned} P &= \{x \in \mathcal{N} : x \text{ is a prime number}\} \\ \mathcal{R} &= \{x : x \text{ is a real number}\} \end{aligned}$$

Approach (2) can be generalised to a general axiom of *comprehension*, which constructs sets by taking all elements of a set which satisfies some property $P(x)$. In fact, you have seen a similar construct in Haskell: the construction

```
[x |  x <- S, p x]
```

denotes a list of terms `x` such that `x` is in the list `S` and the property `p(x)` holds for some predicate `p`. However, a completely unrestricted use of comprehension can cause problems:

> Russel's paradox: the construction $R = \{X : X \text{is a set} \wedge X \notin X\}$ is not a set[1].

Assume R is a set. If $R \in R$, then by the construction of R it follows that $R \notin R$ which is impossible. If $R \notin R$, then by the construction of R it follows that $R \in R$ and again we have a contradiction. Hence, the assumption must be wrong and R cannot be a set. It is possible to remove this sort of paradox using *axiomatic set theory*, which is a very formal definition of set theory. This definition is beyond the scope of this course, and any 'normal' sets you are likely to construct will not encounter this problem.

2.2.1 Basic Set Constructors

We describe set constructors which build new sets from old. In each case. the resulting sets can be assumed to be well-defined as long as the original sets are well-defined.

DEFINITION 2.5 (COMBINING SETS)
Let A and B be any sets. We may construct the following sets:

Set Union	$A \cup B = \{x : x \in A \vee x \in B\}$
Set Intersection	$A \cap B = \{x : x \in A \wedge x \in B\}$
Difference	$A - B = \{x : x \in A \wedge x \notin B\}$
Symmetric difference	$A \triangle B = (A - B) \cup (B - A)$

[1] A colloquial rendition of this paradox is: 'In a certain town, Kevin the barber shaves all those and only those who do not shave themselves. Who shaves the barber?' The riddle has no good answer.

For example, let $A = \{1, 3, 5, 7, 9\}$ and $B = \{3, 5, 6, 10, 11\}$. Then

$$\begin{aligned} A \cup B &= \{1, 3, 5, 6, 7, 9, 10, 11\} \\ A \cap B &= \{3, 5\} \\ A - B &= \{1, 7, 9\} \\ A \triangle B &= \{1, 7, 9, 6, 10, 11\} \end{aligned}$$

It is often helpful to illustrate these combinations of sets using *Venn diagrams*[2].

2.2.2 Properties of Operators

In this section, we investigate certain equalities between sets constructed from our set-theoretic operations.

PROPOSITION 2.6 (PROPERTIES OF OPERATORS)
Let A, B and C be arbitrary sets. They satisfy the following properties:

Commutativity	**Idempotence**
$A \cup B = B \cup A$	$A \cup A = A$
$A \cap B = B \cap A$	$A \cap A = A$
Associativity	**Empty set**
$A \cup (B \cup C) = (A \cup B) \cup C$	$A \cup \emptyset = A$
$A \cap (B \cap C) = (A \cap B) \cap C$	$A \cap \emptyset = \emptyset$
Distributivity	**Absorption**
$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$	$A \cup (A \cap B) = A$
$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$	$A \cap (A \cup B) = A$

Proof We will just look at the first distributivity equality. Some of the other cases will be set as exercises. Draw a Venn diagram to give *some* evidence that the property is indeed true. **It is not a proof!** Let A, B and C be

[2] I will check whether you have been taught Venn diagrams. If not, we will go over them during lectures.

arbitrary sets. The proof is given by:

$$\begin{aligned} A \cup (B \cap C) &= \{x : x \in A \vee x \in (B \cap C)\} \\ &= \{x : x \in A \vee (x \in B \wedge x \in C)\} \\ &= \{x : (x \in A \vee x \in B) \wedge (x \in A \vee x \in C)\} \\ &= \{x : (x \in A \cup B) \wedge (x \in A \cup C)\} \\ &= \{x : x \in (A \cup B) \cap (A \cup C)\} \end{aligned}$$

Notice that this proof depends on the distributivity of the logical connective $\vee$ over $\wedge$ introduced in the logic course.

This style of proof can be confusing to those who are unsure of the logical notation. We can also give another style of proof, which is more wordy but basically reasons in the same way. Let A, B and C be arbitrary sets. We will prove the following two results:

1. $A \cup (B \cap C) \subseteq (A \cup B) \cap (A \cup C)$
2. $(A \cup B) \cap (A \cup C) \subseteq A \cup (B \cap C)$

To prove part 1, assume $x \in A \cup (B \cap C)$ for an arbitrary element x. By the definition of set union, this means that $x \in A$ or $x \in B \cap C$. By the definition of set intersection, this means that either $x \in A$, or x is in both B and C. By the distributivity of 'or' over 'and', it follows that $x \in A$ or $x \in B$, and also that $x \in A$ or $x \in C$. This means that $x \in A \cup B$ and $x \in A \cup C$, and hence $x \in (A \cup B) \cap (A \cup C)$. We have shown that $x \in A \cup (B \cap C)$ implies $x \in (A \cup B) \cap (A \cup C)$.

To prove part 2, we must prove that $x \in (A \cup B) \cap (A \cup C)$ implies $x \in A \cup (B \cap C)$. In this case the proof is simple, since it just follows the above proof in reverse. The details are left as an exercise. □

The above proof is an example of the generality often required to prove a property about sets: it uses arbitrary sets and arbitrary elements of such a set. In contrast, to show that a property is false, it is enough to find one counter-example. Such counter-examples should be as simple as possible, to illustrate that a statement is not true with minimum effort to the reader.

PROPOSITION 2.7
The following statements are not true:

1. $A \cup (B \cap C) = (A \cap B) \cup C$;
2. $A \cup (B \cap C) = (A \cap B) \cup (A \cap C)$.

Proof A simple counter-example to part 1 is $A = \{a\}, B = \{b\}, C = \{c\}$, where a, b, c are different objects. In this example, $B \cap C = \emptyset$ and $A \cup (B \cap C) = \{a\}$ whereas $A \cap B = \emptyset$ and $(A \cap B) \cup C = \{c\}$. A simple counter-example to part 2 is $A = \{a\}, B = C = \{b\}$, where again a, b are different. □

Question: Under what conditions are $A \cup (B \cap C)$ and $(A \cap B) \cup C$ equal?
Answer: It is simple to see the solution by drawing the Venn diagrams. The sets are equal when $A - (B \cup C) = \emptyset$ and $C - (A \cup B) = \emptyset$.

2.2.3 Size of Finite Sets

In this section, we begin to explore the number of elements in a *finite* set. In section 4.6, we will learn how to talk about the number of elements in an *infinite* set.

DEFINITION 2.8 (CARDINALITY)
Let A be a finite set. The *cardinality* of A, written $|A|$, is the number of distinct elements contained in A.

Notice the similarity between this definition and the length function over lists in Haskell. Here are some examples:

$$\begin{aligned} |\{a, e, i, o, u\}| &= 5 \\ |\{a, a, b, c\}| &= 3 \\ |\emptyset| &= 0 \\ |\mathcal{N}| &= \text{ undefined for now} \end{aligned}$$

Fact Let A and B be finite sets. Then $|A \cup B| = |A| + |B| - |A \cap B|$.

Informal proof The number $|A| + |B|$ counts the elements of $A \cap B$ twice, so we subtract $A \cap B$ to obtain the result. A consequence of this proposition is that, if A and B are disjoint sets, then $|A \cup B| = |A| + |B|$.

2.2.4 Introducing Power Sets

In definition 2.2, we defined a notion of a *subset* of a set. The set of all subsets of A is called the *power set*[3] of A.

DEFINITION 2.9 (POWER SET)
Let A be any set. Then the *power set* of A, written $\mathcal{P}(A)$, is $\{X : X \subseteq A\}$

[3]The name might be due to the cardinality result in proposition 2.10. At least this explanation helps to remember the result!

Consider the Haskell function:

```
h' :: [Int] -> [Int] -> [Int]
h' xs rs = [ x | (x,r) <- zip xs rs, r > x]
```

Given the list `xs`, we may obtain any sublist as the output list depending on the list `rs`. Constructing the list of possible output lists associated with list `xs` is analogous to the power set constructor.

Examples of power sets include:

$$\begin{aligned} \mathcal{P}(\{a,b\}) &= \{\emptyset,\{a\},\{b\},\{a,b\}\} \\ \mathcal{P}(\emptyset) &= \{\emptyset\} \\ \mathcal{P}(\mathcal{N}) &= \{\emptyset,\{1\},\{2\},\ldots,\{1,2\},\{1,3\},\ldots,\{2,1\},\{2,2\},\ldots\} \end{aligned}$$

Let A be an arbitrary finite set. One way to list all the elements of $\mathcal{P}(A)$ is to start with $\emptyset$, then add the sets taking one element of A at a time, then the sets talking two elements from A at a time, and so on until the whole set A is added, and $\mathcal{P}(A)$ is complete.

PROPOSITION 2.10
Let A be a finite set with $|A| = n$. Then $|\mathcal{P}(A)| = 2^n$.

Proof This statement is true, but you may need some convincing. If so, test the proposition on an example such as $\mathcal{P}(\{a,b\})$. Here is one proof, which you do not need to remember. Consider an arbitrary set $A = \{a_1,\ldots,a_n\}$. We form a subset X of A by taking each element a_i in turn and deciding whether or not to include it in X. This gives us n independent choices between two possibilities: in X or out. The number of different subsets we can form is therefore 2^n. An alternative way of explaining this is to assign a 0 or 1 to all the elements a_i. Each subset corresponds to a unique binary number with n digits. There are 2^n such posibilities.

Another proof will be given in the reasoning course next term, using the so-called 'induction principle'. □

2.2.5 Introducing Products

The last set construct we consider is the product of two (or arbitrary n) sets. This constructor forms an essential part of the definition of relation discussed in the next section. If we want to describe the relationship 'John loves Mary', then we require a way of talking about John and Mary at the

same time. We do this using an *ordered pair*. An ordered pair (a, b) is a pair of objects a and b where the order in which a and b are written matters. For any objects a, b, c, d, we have $(a, b) = (c, d)$ if and only if $a = c$ and $b = d$. In our example, we have the pair (John, Mary) in the 'loves' relation, but not necessarily the pair (Mary, John) since the love might be unrequited. Hence, the order of the pair is important. The product constructor allows us to collect the ordered pairs together in one set.

DEFINITION 2.11 (CARTESIAN/BINARY PRODUCT)
Let A and B be arbitrary sets. The *Cartesian (or binary) product* of A and B, written $A \times B$, is $\{(a, b) : a \in A \wedge b \in B\}$. We sometimes write A^2 instead of $A \times A$. [Do not confuse the binary relation $\times$ on sets with the standard multiplication $\times$ on numbers.]

Simple examples of Cartesian products include:

1. the coordinate system of real numbers $\mathcal{R}^2$: points are described by their coordinates (x, y);
2. computer marriage bureau: let M be the set of men registered and W the set of women, then the set of all possible matches is $M \times W$;
3. products are analogous to the product types of Haskell: for instance, (Int, Char) is Haskell's notation for the product Int $\times$ Char.

Fact Let A and B be finite sets. Then $|A \times B| = |A| \times |B|$.

Proof If you are uncertain about whether this fact is true, explore examples such as $\{a, b\} \times \{1, 2, 3\}$ and $\emptyset \times \{a, b\}$. We give an informal proof, which you do not need to remember. Suppose that A and B are arbitrary sets with $A = \{a_1, \ldots, a_m\}$ and $B = \{b_1, \ldots, b_n\}$. Draw a table with m rows and n columns of the members of $A \times B$:

$$\begin{array}{llll} (a_1, b_1) & (a_1, b_2) & \ldots \\ (a_2, b_1) & (a_2, b_2) & \ldots \\ \ldots \end{array}$$

Such a table has $m \times n$ entries. □

We can extend this definition of Cartesian product to the n-ary case.

DEFINITION 2.12 (n-ARY PRODUCT)
1. For any $n \geq 1$, an *n-tuple* is a sequence $(a_1, \ldots, a_n)$ of n objects where the order of the a_i matter.

2. Let $A_1, \ldots, A_n$ be arbitrary sets. The *n-ary product* of the A_i, written $A_1 \times \ldots \times A_n$ or $\bigcup_{i=1}^{n} A_i$, is $\{(a_1, \ldots, a_n) : a_i \in A_i \text{ for } 1 \leq i \leq n\}$.

The n-ary product of As is written A^n, with A^2 corresponding to the Cartesian product introduced in definition 2.11. The following examples are simple examples of n-ary products.

1. The three dimensional space of real numbers $\mathcal{R}^3$.

2. The set timetable = day $\times$ time $\times$ room $\times$ courseno: a typical element of this set is (Wednesday, 11.00, 308, 140). In Haskell notation, this timetable example can be given by:

```
type Day = String
type Time = (Int, Int)
type Room = Int
type CourseNo = Int
type Timetable  = (Day, Time, Room, CourseNo)

(Wednesday, (11,00), 308, 140) :: Timetable
```

3. *Record types* are also similar to n-ary products. Suppose we wish to have a database which stores information about people. An array will be unsuitable, since information such as height, age, colour of eyes, date of birth, will be of different types. In many procedural and object-oriented languages, we can instead define

```
Person = RECORD
    who : Name;
    height : Real;
    age : [0...120];
    eyeColour : Colour;
    dateOfBirth : Date
END
```

   This record is like a Haskell type augmented with *projector* functions:

```
type Name = String
type Colour = String
type Date = (Int, Int)
type Person = (Name, Float, Int, Colour, Date)
```

```
who :: Person -> Name
height :: Person -> Float
age :: Person -> Int
eyeColour :: Person -> Colour
dateOfBirth :: Person -> Date

height (_, h, _, _, _) = h
...
```

Just like products, records can be nested, so that in the above example we might have

Date = RECORD
    day : [1...31];
    month : [1...12];
    year : [1900...1990]
END

Fact Let A_i be finite sets for each $1 \leq i \leq n$. Then $|A_1 \times \ldots \times A_n| = |A_1| \times \ldots \times |A_n|$. This fact can be simply proved by the 'induction principle', introduced next term.

Notice that we can now form the product of three sets in three different ways:

$$A \times B \times C \qquad (A \times B) \times C \qquad A \times (B \times C)$$

These sets are all different, so the set-operator $\times$ is not associative [whereas multiplication $\times$ on numbers is associative]. There is however a clear correspondence between the elements (a, b, c) and $((a, b), c)$ and $(a, (b, c))$, and so these sets are in some sense equivalent. We make this intuition precise later in the course. This phenomenon also occurs with Haskell types. A natural example is

```
type Name = String
type Firstname = String
type Secondname = String
type Surname = String

Name1 ::= (FirstName, SecondName, Surname)
```

```
Forenames ::= (FirstName, SecondName)
Name2 ::= (Forenames, Surname)
```

3 Relations

We wish to capture the concept of objects being *related*: for example, John *loves* Mary; $2 < 5$; two programs P and Q are 'equal'. Such properties can be expressed in logic using relations on atomic terms, as you have seen in the logic course. Here we define relations as special sets. For instance, assume the universal set People of all people. We form a set *loves* consisting of all ordered pairs of people such that the first loves the second:

$$\mathsf{loves} = \{(x, y) : x, y \in \mathsf{People} \wedge x \text{ loves } y\}$$

Thus $\mathsf{loves} \subseteq \mathsf{People} \times \mathsf{People}$.

3.1 Introducing Relations

DEFINITION 3.1 (BINARY RELATIONS)
A *binary relation* between (arbitrary) sets A and B is a subset of the binary product $A \times B$.

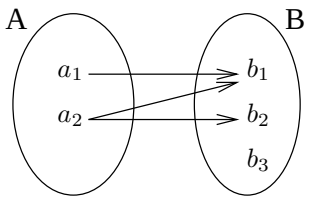
We use $R, S, \ldots$ to range over relations. If $R \subseteq A_1 \times A_2$, we say that R has *type* $A_1 \times A_2$. If $R \subseteq A \times A$, we sometimes just say that R is a binary relation on A. Instead of $(a_1, a_2) \in R$, we often write $R(a_1, a_2)$; for example, we use the logical notation $\mathsf{loves}(x, y)$ rather than $(x, y) \in \mathsf{loves}$. We sometimes write $a\,R\,b$ instead of $R(a, b)$; for example, $x \,\mathsf{loves}\, y$ or $2 < 5$ or a 'f' b in Haskell.

In general, there will be many relations on any set. A relation does not have to be meaningful; any subset of a Cartesian product is a relation. For example, for $A = \{a, b\}$, there are sixteen relations on A:

$\emptyset$	$\{(a, b), (b, a)\}$
$\{(a, a)\}$	$\{(a, b), (b, b)\}$
$\{(a, b)\}$	$\{(b, a), (b, b)\}$
$\{(b, a)\}$	$\{(a, a), (a, b), (b, a)\}$
$\{(b, b)\}$	$\{(a, a), (a, b), (b, b)\}$
$\{(a, a), (a, b)\}$	$\{(a, a), (b, a), (b, b)\}$
$\{(a, a), (b, a)\}$	$\{(a, b), (b, a), (b, b)\}$
$\{(a, a), (b, b)\}$	$\{(a, a), (a, b), (b, a), (b, b)\}$

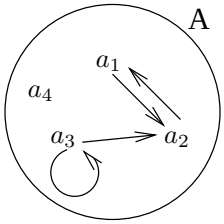
However, listing ordered pairs can get tedious and hard to follow. For binary relations $R \subseteq A \times B$, we have several other representations.

1. (Diagram) Let $A = \{a_1, a_2\}$, $B = \{(b_1, b_2, b_3\}$, $R = \{(a_1, b_1), (a_2, b_1), (a_2, b_2)\}$. We can represent R by the following diagram



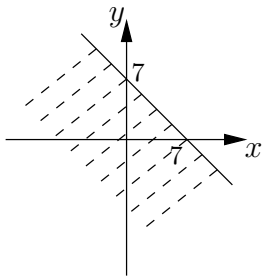
   You should remember this pictorial representation. [We sometimes remove the boundary circles when it is clear which elements belong to which set.]

2. (Directed Graph) In the case where R is a binary relation on A we can also use a *directed graph*, which consists of a set of nodes corresponding to the elements in A, joined by arrowed lines indicating the relationship between the elements. For example, let $A = \{a_1, a_2, a_3, a_4\}$ and $R = \{(a_1, a_2), (a_2, a_1), (a_3, a_2), (a_3, a_3)\}$. The directed graph of this relation is



   Notice that the direction of the arrows matters. It is, of course, still possible to use a diagram where the source and target sets are drawn separately as in 1. You should also remember this formulation. [Again, we sometimes omit the boundary circle.] Directed graphs will be studied further in Discrete Maths 2.

3. (Special representations) We have invented special ways of drawing certain important relations. For example, we can represent a relation on $\mathcal{R}^2$ as an area in the plane. The following diagram represents the relation R defined by $x\,R\,y$ if and only if $x + y \leq 7$.



4. (Matrix) Suppose that $A = \{a_1, a_2, \ldots, a_m\}$ and $B = \{b_1, \ldots, b_n\}$. We can represent R by an $m \times n$ matrix M of booleans (T, F), where recall from the logic course that T stands for True and F for False. For $i = 1, \ldots, m$ and $j = 1, \ldots, n$, define

$$\begin{aligned} \mathsf{M}(i,j) &= \mathsf{T}, \quad a_i\,R\,b_j \\ &= \mathsf{F}, \quad \text{otherwise} \end{aligned}$$

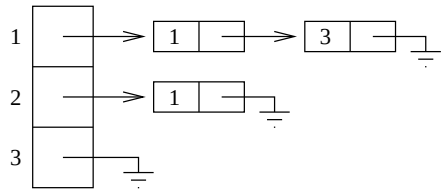
where $\mathsf{M}(i,j)$ is the usual notation for the ith row and jth column of the matrix. For example, if $A = \{a_1, a_2\}$, $B = \{b_1, b_2, b_3\}$ and $R = \{(a_1, b_1), (a_2, b_1), (a_2, b_2)\}$ as before, then the matrix is

$$\begin{pmatrix} \mathsf{T} & \mathsf{F} & \mathsf{F} \\ \mathsf{T} & \mathsf{T} & \mathsf{F} \end{pmatrix}$$

It is also common to use the elements $0, 1$ instead of F, T. You do not need to remember this formulation, although past exam questions have asked questions about this representation.

5. (Implementation) On a computer, we can store a relation using an array. This allows random access and easy manipulation, but can be expensive in space if the relation is much smaller that $A \times B$. With a *sparse* relation, where there are not many ordered pairs, an alternative approach is to use an array of linked lists, called an *adjacency list*. For example, consider the binary relation $R = \{(1,1), (1,3), (2,1)\}$ on set

$\{1, 2, 3\}$. This relation only has 3 of the possible 9 ordered pairs. We create an array of three pointers, one for each element of $\{1, 2, 3\}$, and list for each element which other elements it is related to:



Just as for products, we can extend the definition of a binary relation to an arbitrary n-ary relation.

DEFINITION 3.2
A *n-ary relation* between sets $A_1, \ldots, A_n$ is a subset of a n-ary product $A_1 \times \ldots \times A_n$. The definition of a 2-ary relation is the same as that of a binary relation given in definition 2.11. A *unary relation*, or *predicate*, over set A is a 1-ary relation: that is, a subset of A.

EXAMPLE 3.3
1. The set $\{x \in \mathcal{N} : x \text{ is prime}\}$ is a unary relation on $\mathcal{N}$.

2. The set $\{(x, y, z) \in \mathcal{R}^3 : \sqrt{x^2 + y^2 + z^2} = 1\}$ is a 3-ary relation on the real numbers, which describes the surface of the unary sphere with centre $(0, 0, 0)$.

3.2 Constructing relations

Just as for sets, we may construct new relations from old. We just give the definitions for binary relations. it is easy to extend the definitions to the n-ary case.

DEFINITION 3.4 (BASIC RELATION OPERATORS)
Let $R, S \subseteq A_1 \times A_2$. Define the relations $R \cup S$, $R \cap S$ and $\overline{R}$, all with type $A_1 \times A_2$, by

1. (Relation Union) $(a_1, a_2) \in R \cup S$ iff $(a_1, a_2) \in R$ or $(a_1, a_2) \in S$;

2. (Relation Intersection) $(a_1, a_2) \in R \cap S$ if and only if $(a_1, a_2) \in R$ and $(a_1, a_2) \in S$;

3. (Relation Complement) $(a_1, a_2) \in \overline{R}$ if and only if $(a_1, a_2) \in A_1 \times A_2$ and $(a_1, a_2) \notin R$.

EXAMPLE 3.5
Let R and S be binary relations on $\{1, 2, 3, 4\}$ such that

$$\begin{aligned} R &= \{(1,2),(2,3),(3,4),(4,1)\} \\ S &= \{(1,2),(2,1),(3,4),(4,3)\} \end{aligned}$$

We may construct the following relations:

$$\begin{aligned} R \cup S &= \{(1,2),(2,3),(3,4),(4,1),(2,1),(4,3)\} \\ R \cap S &= \{(1,2),(3,4)\} \\ \overline{R} &= \{(1,1),(1,3),(1,4),(2,1),(2,2),(2,4),(3,1),(3,2),(3,3),(4,2), \\ &\quad (4,3),(4,4)\} \end{aligned}$$

We have overloaded notation: $R \cup S$ and $R \cap S$ denotes relation union and intersection respectively when R and S are viewed as relations, and set union and intersection when viewed as sets. Notice that to form a relation union or intersection, the relations must be of the same type. In contrast, we can form the union and intersection of arbitrary sets. It should be clear from the context which interpretation we intend.

DEFINITION 3.6 (IDENTITY RELATION)
Given any set S, the *identity* on A, written id_A, is a binary relation on A defined by $\mathsf{id}_A = \{(a, a) : a \in A\}$.

DEFINITION 3.7 (INVERSE RELATION)
Let $R \in A \times B$ denote an arbitrary binary relation. The *inverse* of R, written R^{-1}, is defined by $a\, R^{-1}\, b$ if and only if $b\, R\, a$.

Inverse should not be confused with complement: for example, the inverse of 'is a parent of' is 'is the child of'; if z is the cousin of y, then z is in the complement of 'is a parent of', but not the inverse. Using the R from example 3.5, the inverse $R^{-1} = \{(2,1),(3,2),(4,3),(1,4)\}$. If we take the inverse of the inverse of a relation, we recover the original relation: $(R^{-1})^{-1} = R$.

DEFINITION 3.8 (COMPOSITION OF RELATIONS)
Given $R \subseteq A \times B$ and $S \subseteq B \times C$, then the *composition* of R with S, written $R \circ S$, is defined by

$$a\, R{\circ}S\, c \text{ if and only if } \exists b \in B.\, (a\, R\, b \wedge b\, S\, c)$$

The notation $R \circ S$ may be read as 'R composed with S' or 'R circle S'. The relation $R \circ S$ is only defined if the types of R and S match up. For example, we can define the set $\mathsf{grandparent} = \mathsf{parent} \circ \mathsf{parent}$ by:

x grandparent of y iff $\exists z.$ (x parent of z) $\wedge$ (z parent of y).

Using the R and S from example 3.5, we have

$$\begin{aligned} R \circ S &= \{(1,1),(2,4),(3,3),(4,2)\} \\ S \circ R &= \{(1,3),(2,2),(3,1),(4,4)\} \end{aligned}$$

Contrast this notation for relational composition with the Haskell notation for functional composition:

```
(g.f) x = g ( f x)
```

With the Haskell notation, the functional composition `g . f` reads '`f` followed by g', whereas the relational composition $R \circ S$ reads 'R followed by S'. Very confusing!

3.3 Equalities between Relations

Recall from proposition 2.5 that we can prove certain equalities between sets constructed from the set operations. We can also similar equalities between relations constructed from the operations on relations.

PROPOSITION 3.9

1. If $R \subseteq A \times B$, then $\mathsf{id}_A \circ R = R = R \circ \mathsf{id}_B$.
2. $\circ$ is associative: that is, for arbitrary relations $R \subseteq A \times B$ and $S \subseteq B \times C$ and $T \subseteq C \times D$, then

$$R \circ (S \circ T) = (R \circ S) \circ T.$$

Proof The proof of part 1 is simple and left as an exercise. We prove part 2. Let R, S and T be relations specified in the proposition, and let (x,u) be an arbitrary member of $(R \circ S) \circ T$. We show that $(x,u) \in R \circ (S \circ T)$ using the following reasoning:

$$\begin{aligned} x\,(R \circ S) \circ T\,u &\Rightarrow \exists z.\ x\,(R \circ S)\,z \wedge z\,T\,u \\ &\Rightarrow \exists z.\ (\exists y.\ x\,R\,y \wedge y\,S\,z) \wedge z\,T\,u \\ &\Rightarrow \exists z, y.\ (x\,R\,y \wedge y\,S\,z \wedge z\,T\,u) \\ &\Rightarrow \exists y.\ (x\,R\,y) \wedge (\exists z.\ y\,S\,z \wedge z\,T\,u) \\ &\Rightarrow \exists y.\ (x\,R\,y) \wedge (y\,S \circ T\,u) \\ &\Rightarrow x\,R \circ (S \circ T)\,u \end{aligned}$$

We have shown that $(R \circ S) \circ T \subseteq R \circ (S \circ T)$. The reverse direction showing that $R \circ (S \circ T) \subseteq (R \circ S) \circ T$ can be proved in a similar way. □

PROPOSITION 3.10
Let R and S be arbitrary binary relations on A. In general

1. $R \neq R^{-1}$;
2. composition is not commutative: that is, $R \circ S \neq S \circ R$;
3. $R \circ R^{-1} \neq \mathsf{id}_A$.

Proof Just as for proposition 2.7, the way to prove that a property does not hold is to provide a counter-example. A counter-example to part 1 is the relation $R = \{(a, b)\} \subseteq \{a, b\} \times \{a, b\}$. Then $R^{-1} = \{(b, a)\}$ which is plainly different from R. To show that composition is not commutative, we must find R, S such that $R \circ S \neq S \circ R$. Let $A = B = \{a, b\}$, $R = \{(a, a)\}$ and $S = \{(a, b)\}$. Then $R \circ S = \{(a, b)\}$ but $S \circ R = \emptyset$. Part 3 is left as an exercise.

3.4 Application to Relational Databases

A relational database is a collection of relations. We describe further operations on relations which are key operations used in relational databases. [We only deal with the static aspects of databases, not concerning ourselves with updating and maintaining integrity.] Consider the example of a university registry database, which has a relation Student storing the students' names, addresses and examination numbers. It is usual to represent such a database relation as a table:

name	address	number
...	...	...
Brown, B	5 Lawn Rd.	105
Jackson, B.	1 Oak Dr.	167
Smith, J.	9 Elm St.	156
Walker, S.	4 Ash Gr.	189
...	...	...

Each tuple of the relation corresponds to a row in the table. The records in a database, in this case name, address, number, are called the *attributes* of the relation; each attribute corresponds to a column. Associated with each

attribute is a set (or *domain*) from which it takes its values. It is clear that these database relations are just the same as the n-ary relations we have been studying. In our example, we may write

$$\mathsf{Student} \subseteq \mathsf{name\_set} \times \mathsf{address\_set} \times \mathsf{number\_set}$$

using an obvious notation for the sets associated with each attribute.

Suppose that the registry database has another relation, called Exam, which records the results for students taking the compilers exam. It has attributes number and grade. A table for Exam might look like

number	grade
...	...
105	A
156	A
189	C
...	...

Notice that the relations Student and Exam share an attribute, namely number. We can combine the two relations using an operation called *join* to get a new relation, which we call Student_Exam, which merges the two relations on their common attribute:

name	address	number	grade
...	...	...	...
Brown, B	5 Lawn Rd.	105	A
Smith, J	9 Elm St.	156	A
Walker, S	4 Ash Gr.	189	C
...	...	...	...

Notice that candidate 167 did not sit the exam, and so therefore does not appear in the join. We can define this join operation quite easily using our logical notation:

$$\mathsf{Student\_Exam(n,a,no,g)} \Leftrightarrow \mathsf{Student(n,a,no)} \wedge \mathsf{Exam\ (no,g)}$$

In the language of database theory, it is usually given a more readable form, such as

join Student **and** Exam **over** number

We might wish to modify Student_Exam by hiding the number information to get a new relation Results. This can be done by the operation *projection*, to yield the following result:

name	address	grade
...	...	...
Brown, B	5 Lawn Rd.	A
Smith, J	9 Elm St.	A
Walker, S	4 Ash Gr.	C
...	...	...

In our logical notation, we may write:

Results(n,a,g) if and only if $\exists$ no. Student_Name(n,a,no,g)

In database notation, this would be written in the style

project Student_Exam **over** (name, address, grade)

Notice that Results is obtained from Student and Exam by a sort of generalised composition. In fact, composition of binary relations can be constructed by a join followed by a projection.

Another operation we might wish to do is to *select* a part of a relation table which is of interest. Suppose in our registry example we wish to have the names of those students who should be considered for a prize, and so we select those candidates who got A in the exam. Starting from the relation Results, we obtain a relation A-Results:

name	address	grade
...	...	...
Brown, B	5 Lawn Rd.	A
Smith, J	9 Elm St.	A
...	...	...

In logical notation, we could write

A-Results(n,a,g) if and only if Results(n,a,g) $\wedge$ g = 'A'

In database notation we have

select results **where** grade = 'A'

The relation A-Results gives us the names we want, but we could reduce further to get the relation PrizeCands with the single attribute:

name
...
Brown, B
Smith, J
...

We have introduced three database operations—join, projection, selection—and have seen how each operation has a counterpart in our formalism. More details about relational databases will be given in the database course in the second term. Relations will also be used in the second half of the Declarative Programming Course: Logic Programming.

3.5 Properties of Relations

From section 1, recall our definition of two Haskell functions being equivalent:

> Two Haskell functions $\mathtt{f} : \mathtt{A} \to \mathtt{B}$ and $\mathtt{g} : \mathtt{A} \to \mathtt{B}$ are equivalent, written $\mathtt{f} = \mathtt{g}$, if and only if, for all terms $\mathtt{a}$ in type $\mathtt{A}$, then if $\mathtt{f\,a}$ terminates then $\mathtt{g\,a}$ terminates and $\mathtt{f\,a} = \mathtt{g\,a}$, and if $\mathtt{f\,a}$ does not terminate then $\mathtt{g\,a}$ does not terminate.

It is simple to show that the following properties are satisfied for this definition of equivalence between Haskell functions:

- (reflexivity) $\forall \mathtt{f}.\ \mathtt{f} = \mathtt{f}$;
- (symmetry) $\forall \mathtt{f}_1, \mathtt{f}_2.\ \mathtt{f}_1 = \mathtt{f}_2 \Rightarrow \mathtt{f}_2 = \mathtt{f}_1$;
- (transitivity) $\forall \mathtt{f}_1, \mathtt{f}_2, \mathtt{f}_3.\ \mathtt{f}_1 = \mathtt{f}_2 \wedge \mathtt{f}_2 = \mathtt{f}_3 \Rightarrow \mathtt{f}_1 = \mathtt{f}_3$.

We give universal definitions for the properties just described. A relation may or may not satisfy such properties.

DEFINITION 3.11
Let R be a binary relation on A. Then

1. R is *reflexive* if and only if $\forall x \in A.\ x\,R\,x$;
2. R is *symmetric* if and only if $\forall x, y \in A.\ x\,R\,y \Leftrightarrow y\,R\,x$;
3. R is *transitive* if and only if $\forall x, y, z \in A.\ x\,R\,y \wedge y\,R\,z \Rightarrow x\,R\,z$.

Relations with these properties occur naturally: the equality relation on sets is reflexive, symmetric and transitive; the relations $\leq$ on numbers and $\subseteq$ on sets are reflexive and transitive, but not symmetric; and the relation $<$ on numbers is transitive, but not reflexive nor symmetric.

Another way to define these relations is in terms of the operations on relations introduced in the previous section.

PROPOSITION 3.12
let R be a binary relation on A.

1. The relation R is reflexive if and only if $\mathsf{id}_A \subseteq R$.
2. The relation R is symmetric if and only if $R = R^{-1}$.
3. The relation R is transitive if and only if $R \circ R \subseteq R$.

Proof The proof is easy and is left as an exercise.

3.6 Equivalence Relations

We think of an equivalence relation as a weak equality: $a\,R\,b$ means that a and b are in some sense indistinguishable. For example, imagine that we have a set of programs and we have various demands to make of them: for example, we might require that the programs

- always terminate;
- cost less than a hundred pounds;
- compute π to 100 decimal places;
- ...

Even though two programs are not equal, they can satisfy the same demands and so be 'equal enough' for our purposes. In such a case, we say that two programs are *equivalent*.

DEFINITION 3.13
Let A be a set and R a binary relation on A. The relation R is an *equivalence relation* if and only if R is reflexive, symmetric and transitive. We sometimes just say that R is an *equivalence*.

Examples

1. Given $n \in \mathcal{N}$, the binary relation R on $\mathcal{Z}$ defined by $a\,R\,b$ if and only if n divides into $(b-a)$ is an equivalence relation.

2. The binary relation S on the set $\mathcal{Z} \times \mathcal{N}$ defined by $(z_1, n_1)\,S\,(z_2, n_2)$ if and only if $z_1 \times n_2 = z_2 \times n_1$ is an equivalence.

3. Let A be any set. Then the identity relation $\mathsf{id}_A : A \times A$ is an equivalence relation.

4. Given a set Student and a map $\mathsf{age} : \mathsf{Student} \to \mathcal{N}$, the binary relation sameage on Student defined by

$$s_1 \text{ sameage } s_2 \text{ if and only if } \mathsf{age}(s_1) = \mathsf{age}(s_2)$$

   is an equivalence.

5. In definition 4.22, we define the relation $\sim$ between sets, which characterises when (finite and infinite) sets have the same number of elements. Proposition 4.23 shows that $\sim$ is an equivalence relation.

6. The logical equivalence between formulae, given by $A \equiv B$ if and only if $\vdash A \leftrightarrow B$, is an equivalence.

The above examples suggest that equivalence relations lead to natural partitions of the elements into disjoint subsets. The elements in these subsets are related and equivalent to each other.

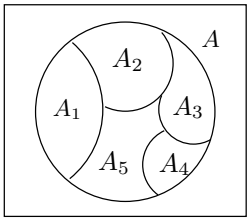
DEFINITION 3.14 (EQUIVALENCE CLASSES)
Let R be an equivalence relation on A. For any $a \in A$, the *equivalence class* of a with respect to R, denoted $[a]_R$, is defined as

$$[a]_R = \{x \in A : a\,R\,x\}.$$

We often write $[a]$ instead of $[a]_R$ when the relation R is apparent.

The set of equivalence classes is sometimes called the quotient set A/R. In examples 1 and 2 just given, $\mathcal{Z}/R$ represents the integers *modulo* n, and $(\mathcal{Z} \times \mathcal{N})/S$ is the usual representation of the rational numbers.

The equivalence classes of a set A can be represented by a Venn diagram: for example,



In this case, there are five equivalence classes, illustrated by the five disjoint subsets. In fact, the equivalence classes always separate the elements into disjoint subsets that cover the whole of the set, as the following proposition states formally.

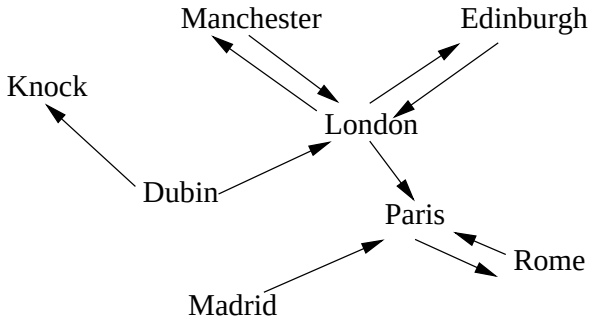
PROPOSITION 3.15
The set of equivalence classes $\{[a] : a \in A\}$ forms a *partition* of A: that is,

- each $[a]$ is non-empty;
- the classes *cover* A: that is, $A = \bigcup_{a \in A} [a]$;
- the classes are disjoint (or equal): $\forall a, b \in A.\, [a] \cap [b] \neq \emptyset \Rightarrow [a] = [b]$.

Proof Given any $a \in A$, then $a\,R\,a$ by reflexivity and so $a \in [a]$. Also $a \in \bigcup_{a \in A} [a]$, and hence the classes cover A. Suppose $[a] \cap [b] \neq \emptyset$, and let $x \in [a] \cap [b]$. This means that $a\,R\,x$ and $b\,R\,x$. It follows that $x\,R\,b$ by symmetry. Given any $v \in [b]$, observe that $b\,R\,v$. Now $a\,R\,x$, $x\,R\,b$ and $b\,R\,v$, so $a\,R\,v$ by transitivity. Therefore $v \in [a]$ and so $[b] \subseteq [a]$. But $[a] \subseteq [b]$ using a similar argument, so $[a] = [b]$. □

3.7 Transitive Closure

Consider the following situation. There are various flights between various cities. For any two cities, we wish to know whether it is possible to fly from one to the other allowing for changes of plane. We can model this by defining a set City of cities and a binary relation R such that $a\,R\,b$ if and only if there is a direct flight from a to b. This relation may be represented as a directed graph with the cities as nodes, as in the following example:



Define the relation R^+ by $a\,R^+\,b$ if and only if there is a trip from a to b. Then clearly $a\,R^+\,b$ if and only if there is some path from a to b in the directed graph. For instance, there is a path from Manchester to Rome, but no path from Rome to Manchester. We would like to calculate R^+ from R. Such a relation is called the *transitive closure* of R, since it is clearly transitive, and is in fact a special relation in the sense that it is the smallest transitive relation containing R.

We can express the relation R^+ in terms of R using relational composition: $a\,R^+\,b$ if and only if there is a path of length n from a to b, for some $n \geq 1$. Another way of making this statement is to define the interim relations R^n: $a\,R^n\,b$ if and only if there is a path of length n from a to b. Another way of defining R^n is

$$
\begin{aligned}
&R^1 = R \\
&R^2 = R \circ R \\
&R^3 = R \circ R^2 = R^2 \circ R, \quad \text{since } \circ \text{ is associative} \\
&\vdots \\
&R^n = R \circ R^{n-1} = R \circ \ldots \circ R, \quad n \text{ times} \\
&\vdots
\end{aligned}
$$

Therefore, we have $a\,R^+\,b$ if and only if $\exists n \geq 1.\ a\,R^n\,b$: moreover

$$R^+ = R \cup R^2 \cup \ldots \cup R^n \cup \ldots = \bigcup_{n \geq 1} R^n$$

There are many other natural examples of transitive closure, such as the following examples:

1. Program modules can import other modules. [You have seen this already in the Haskell course.] They can also depend indirectly on modules via some chain of importation, so that for instance M depends on M' if M imports M'' and M'' imports M'. The relation 'depends' is the transitive closure of the relation 'imports'.

2. Two people are related if one is the parent of the other, if they are married, or if there is a chain of such relationships joining them directly. We can model this by a universal set People, with three relations Married, Parent and Relative. The relation Relative is defined using the transitive closure:

$$\mathsf{Relative} = ((\mathsf{Parent} \cup \mathsf{Parent}^{-1}) \cup \mathsf{Married})^+$$

DEFINITION 3.16 (TRANSITIVE CLOSURE)
Let R be a binary relation on A. The *transitive closure* of R, written $t(R)$ or R^+, is $\bigcup_{n\geq 1} R^n$: that is,

$$R \cup R^2 \cup R^3 \cup R^4 \cup \ldots$$

The transitive closure of a binary relation always exists. To cast more light on R^+, we now examine an alternative way of building the transitive closure. Let R be finite, and imagine that we want to make R transitive in the most 'economical' fashion. If R is already transitive, we need do nothing. Otherwise, there exists $a, b, c \in A$ such that $a\,R\,b\,R\,c$, but *not* $a\,R\,c$. We add the pair (a, c) to the relation.It is now in some sense closer to being transitive. We carry on doing this, until there are no more pairs. We now have a transitive relation. Anything we have added to R was forced upon us by the requirement of transitivity, so we have obtained the smallest possible relation containing R.

Another approach is to see how a computer might compute R^+. In order to calculate R^+, we can compute successively

$$R, R \cup R^2, R \cup R^2 \cup R^3, \ldots$$

In terms of paths, the relation $R \cup R^2 \cup \ldots \cup R^n$ represents all paths of length between 1 and n. But since R^+ is defined as an infinite union it seems that we will have to carry on computing for ever, which will not do. If R is finite however, the process will come to an end at some finite stage because eventually nothing new will be added. Suppose the set on which R is defined has n elements. Then we need not consider paths of length

greater than n since they will involve repeats (visiting the same node of the graph twice). So R^{n+1} is already included in $R \cup R^2 \cup \ldots \cup R^n$ and we need not calculate further as we have found R^+. In fact, we often don't have to go as far as n. In the airline example at the beginning of the section there are 8 cities, but the longest paths without repeats are of length 3. Thus we compute $R \cup R^2 \cup R^3$ and find that $R^4 \subseteq R \cup R^2 \cup R^3$, so that we can stop. We may describe our procedure by the following Kenya-like algorithm where $:=$ denotes assignment:

Input R
$S := R$
$T := R$
$S := R \circ S$
while not $S \subseteq T$ **do**
    $T := T \cup S$
    $S := R \circ S$
od
Output T

In the above algorithm, whenever the while loop is entered, then $S = R^{n+1}$ and $T = R \cup R^2 \cup \ldots \cup R^n$ for $n = 1, 2, 3, \ldots$. There are many ways of improving the algorithm. A very much more efficient method is *Warshall's algorithm*, described in Discrete Maths 2.

It is sometimes useful to 'reverse' the process of finding the transitive closure. In other words, given a transitive relation R, the task is to find a smallest S such that $S^+ = R$. The benefit is that S is smaller, while in some sense having the same information content as R, since R can be reconstructed from S. In general, there can be many solutions for S. We will return to this problem in the easier setting of partial orders in section 5.

4 Functions

You have probably spent a large part of your mathematical education considering mathematical functions in one context or another. In this section, we formalize the notion of mathematical functions as special relations, giving the basic definitions, ways of constructing functions and properties for reasoning about functions. You have also been introduced to Haskell functions, which are examples of the *computable* functions. In future courses, you will learn about these computable functions.

4.1 Introducing Functions

DEFINITION 4.1 (FUNCTIONS)
A *function* f from a set A to a set B, written $f : A \to B$, is a relation $f \subseteq A \times B$ such that *every* element of A is related to *one* element of B; more formally, it is a relation which satisfies the following additional properties:

1. $(a, b_1) \in f \wedge (a, b_2) \in f \Rightarrow b_1 = b_2$;
2. $\forall a \in A. \exists b \in B. (a, b) \in f$.

The set A is called the *domain* of f, and B is the *co-domain* of f. If $a \in A$, then $f(a)$ denotes the unique $b \in B$ such that $(a, b) \in f$. One awkward convention is that, if the domain A is the n-ary product $A_1 \times \ldots \times A_n$, then we often write $f(a_1, \ldots, a_n)$ instead of $f((a_1, \ldots, a_n))$. The intended meaning should be clear from the context. Also, recall the difference between the following two Haskell functions:

```
f :: a -> b -> c -> d
f :: (a,b,c) -> d
```

In definition 4.1, the definition of function can not be curried.

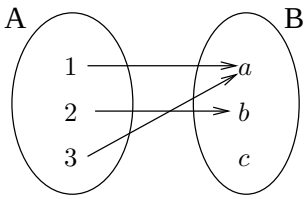
DEFINITION 4.2
Let $f : A \to B$. For any $X \subseteq A$, define the *image* of X under f to be

$$f[X] \stackrel{\text{def}}{=} \{f(a) : a \in X\}$$

The set $f[A]$ of all images of f is called the *image set of* f.

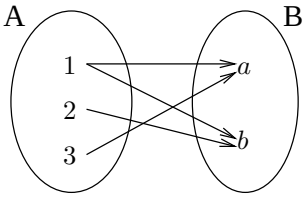
We explore some examples. Since functions are special binary relations, we can use the representations given in section 3 to describe relations. When the domain and co-domain are finite, a useful representation is the *diagram* representation.

1. Let $A = \{1, 2, 3\}$ and $B = \{a, b, c\}$. Let $f \subseteq A \times B$ be defined by $f = \{(1, a), (2, b), (3, a)\}$. Then f is a function, as is clear to see from the diagram:



   The image set of f is $\{a, b\}$. The image of $\{1, 3\}$ under f is $\{a\}$.

2. Let $A = \{1, 2, 3\}$ and $B = \{a, b\}$. Let $f \subseteq A \times B$ be defined by $f = \{(1, a), (1, b), (2, b), (3, a)\}$. This f is not a well-defined function, since one element of A is related to two elements in B as is evident from the diagram:



3. The following are examples of functions with infinite domains and co-domains:

   (a) the function $f : \mathcal{N} \times \mathcal{N} \mapsto \mathcal{N}$ defined by $f(x, y) = x + y$;

   (b) the function $f : \mathcal{N} \mapsto \mathcal{N}$ defined by $f(x) = x^2$;

   (c) the function $f : \mathcal{R} \mapsto \mathcal{R}$ defined by $f(x) = x + 3$.

   The binary relation R on the reals defined by $x\, R\, y$ if and only if $x = y^2$ is not a function, since for example 4 relates to both 2 and -2.

PROPOSITION 4.3 (FINITE CARDINALITY)
Let $A \to B$ denote the set of all functions from A to B, where A and B are finite sets. If $|A| = m$ and $|B| = n$, then $|A \to B| = n^m$.

Sketch proof For each element of A, there are m independent ways of mapping it to B. You do not need to remember this proof. □

4.2 Partial Functions

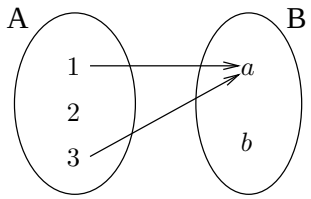
Recall that it is easy to write Haskell functions which return run-time errors on some (or all) arguments. For example, when designing a program P to compute square roots, it is quite reasonable to have P return an error message for negative inputs. We can either regard P as a function which returns the error answer on negative inputs, or we can regard the program as a *partial* function which is undefined on negative arguments.

DEFINITION 4.4
A *partial* function f from a set A to a set B, written $f : A \rightharpoonup B$, is a relation $f \subseteq A \times B$ such that just *some* elements of A are related to unique elements of B; more formally, it is a relation which satisfies:

$$(a, b_1) \in f \wedge (a, b_2) \in f \Rightarrow b_1 = b_2$$

The partial function f is regarded as *undefined* on those elements which do not have an image under f. It is sometimes convenient to refer to this undefined value explicitly as $\perp$ (pronounced *bottom*). A partial function from A to B is the same as a function from A to $(B + \{\perp\})$.

Example The relation $R = \{(1, a), (3, a)\}$ is a partial function:



We can see that not every element in A maps to an element in B.

Example The binary relation R on $\mathcal{R}$ defined by $x\,R\,y$ if and only if $\sqrt{x} = y$ is a partial function. It is not defined when x is negative.

Exercise Let $A \rightharpoonup B$ denote the set of all partial functions from A to B. If $|A| = m$ and $|B| = n$, what is the cardinality of $|A \rightharpoonup B|$?

4.3 Properties of Functions

Recall that we highlighted certain properties of relations, such as reflexivity, symmetry and transitivity. We also highlight certain properties of functions, which will be used to extend our cardinality definition to infinite sets.

DEFINITION 4.5 (PROPERTIES OF FUNCTIONS)
Let $f : A \to B$ be a function. We define the following properties on f:

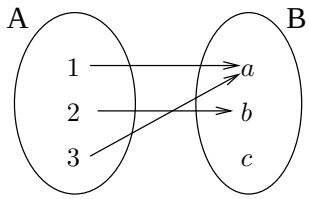
1. f is *onto* (sometimes called *surjective*) if and only if every element of B is in the image of f: that is, $\forall b \in B.\, \exists a \in A.\, f(a) = b$;

2. f is *one-to-one* (sometimes called *injective*) if and only if for each $b \in B$ there is at most one $a \in A$ with $f(a) = b$: that is, $\forall a, a' \in A.\, f(a) = f(a')$ implies $a = a'$;

3. f is a *bijection* if and only if f is *both* one-to-one and onto.

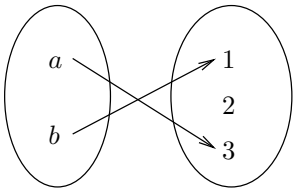
Notice that the definition of an one-to-one function is equivalent to $a_1 \neq a_2 \Rightarrow f(a_1) \neq f(a_2)$, and so an one-to-one function never repeats values.

EXAMPLE 4.6
Let $A = \{1, 2, 3\}$ and $B = \{a, b\}$. The function $f = \{(1, a), (2, b), (3, a)\}$ is onto, but *not* one-to-one, as is immediate from its diagram:



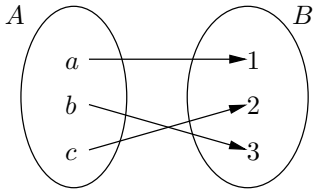
It is not possible to define a one-to-one function from A to B, as there are too many elements in A for them to map uniquely to B.

EXAMPLE 4.7
Let $A = \{a, b\}$ and $B = \{1, 2, 3\}$. The function $f = \{(a, 3), (b, 1)\}$ is one-to-one, but not onto:



It is not possible to define an onto function from A to B in this case, as there are not enough elements in A to map to all the elements of B.

EXAMPLE 4.8
Let $A = \{a, b, c\}$ and $B = \{1, 2, 3\}$. The function $f = \{(a, 1), (b, 3), c, 2)\}$ is bijective:



EXAMPLE 4.9
The function f on natural numbers defined by $f(x, y) = x + y$ is onto but not one-to-one. To prove that f is onto, take an arbitrary $n \in \mathcal{N}$. We must find $(m_1, m_2) \in \mathcal{N} \times \mathcal{N}$ such that $f(m_1, m_2) = n$. This is easy since $f(n, 0) = n + 0 = n$. To show that f is not one-to-one, we need to produce a counter-example. In other words, we must find $(m_1, m_2), (n_1, n_2)$ such that

$(m_1, m_2) \neq (n_1, n_2)$, but $f(m_1, m_2) = f(n_1, n_2)$. There are many possibilities, such as $(1, 0)$ and $(0, 1)$. In fact, since $+$ is commutative, $(m, n), (n, m)$ is a counter-example for any m, n.

EXAMPLE 4.10
The function f on natural numbers defined by $f(x) = x^2$ is one-to-one, but the similar function f on integers is not. The function f on integers defined by $f(x) = x + 1$ is surjective, but the similar function on natural numbers is not.

EXAMPLE 4.11
The function f on the real numbers given by $f(x) = 4x + 3$ is a bijective function. To prove that f is one-to-one, suppose that $f(n_1) = f(n_2)$, which means that $4n_1 + 3 = 4n_2 + 3$. It follows that $4n_1 = 4n_2$, and hence $n_1 = n_2$. To prove that f is onto, let n be an arbitrary real number. We have $f((n - 3)/4) = n$ by definition of f, and hence f is onto. Since f is both one-to-one and onto, it is bijective. Notice that the function f on the *natural* numbers given by $f(x) = x + 3$ is one-to-one but *not* onto, since 2 is not in the image-set of f.

4.4 The Pigeonhole Principle

Suppose that m objects are to be placed in n pigeonholes, where $m > n$. Then some pigeonhole will have to contain more than one object. A similar example is that, if there are at least 367 people in a room, then at least two must share the same birthday. This idea is called the *pigeonhole principle*, due to the illustrative example just given. Let us rephrase the pigeonhole principle in our formal language of functions:

> Let $f : A \to B$ be a function, where A and B are finite. If $|A| > |B|$, then f cannot be a one-to-one function.

Recall example 4.6. We stated that is not possible to define a one-to-one function from $A = \{1, 2, 3\}$ to $B = \{a, b\}$, since A is too big. It is not possible to prove this property directly. Instead, the pigeonhole principle states that we assume that the property is true.

PROPOSITION 4.12
Let A and B be finite sets, let $f : A \to B$ and let $X \subseteq A$. Then $|f[X]| \leq |X|$.

Proof This property is intuitively clear, and can be proved by appealing to the pigeonhole principle. Suppose for contradiction that $|f[X]| > |X|$. Define a place function $p : f[X] \to X$ by

$p(b) =$ some $a \in X$ such that $f(a) = b$.

It does not matter which a we choose, but there will be such an a by definition of $f[X]$. We are placing the members of $f[X]$ in the pigeonholes X. By the pigeonhole principle, some pigeonhole has at least two occupants. In other words, there is some $a \in X$ and $b, b' \in f[X]$ with $p(b) = p(b') = a$. But then $f(a) = b$ and $f(a) = b'$, which cannot happen as f is a function. □

PROPOSITION 4.13
Let A and B be *finite* sets, and let $f : A \to B$.

1. If f is one-to-one, then $|A| \leq |B|$.
2. If f is onto, then $|A| \geq |B|$.
3. If f is a bijection, then $|A| = |B|$.

Proof Part (a) is the contrapositive of the pigeonhole principle. For (b), notice that if f is onto then $f[A] = B$, so that in particular $|f[A]| = |B|$. Also $|A| \geq |f[A]|$ by proposition 4.12. Therefore $|A| \geq |B|$ as required. Finally, part (c) follows from parts (a) and (b). □

4.5 Operations on Functions

Since functions are special relations, we can define the identity, composition and inverse relations of functions. The composition of two functions is always a function. In contrast, we shall see that the inverse relation of a function need not necessarily be a function.

DEFINITION 4.14 (COMPOSITION OF FUNCTIONS)
Let A, B and C be arbitrary sets, and let $f : A \to B$ and $g : B \to C$ be arbitrary functions of these sets. The *composition* of f with g, written $g \circ f : A \to C$, is a function defined by

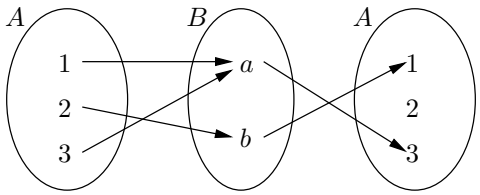
$$g \circ f(a) \stackrel{\text{def}}{=} g(f(a))$$

for every element $a \in A$. In Haskell notation, we would write

```
(g.f) a = g (f a)
```

It is easy to check that $g \circ f$ is indeed a function. Notice that the co-domain of f *must* be the same as the domain of g for the composition to be well-defined.

Example Let $A = \{1, 2, 3\}$, $B = \{a, b, c\}$, $f = \{(1, a), (2, b), (3, a)\}$ and $g = \{(a, 3), (b, 1)\}$. Then $g \circ f = \{(1, 3), (2, 1), (3, 3)\}$:

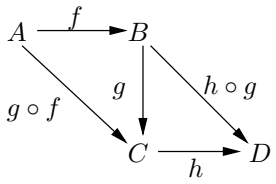


PROPOSITION 4.15 (ASSOCIATIVITY)
Let $f : A \to B$, $g : B \to C$ and $h : C \to D$ be arbitrary sets. Then $h \circ (g \circ f) = (h \circ g) \circ f$.

Proof This result is easily established from the definition of functional composition. Take an arbitrary element $a \in A$. Then

$$(h \circ (g \circ f))(a) = h((g \circ f)(a)) = h(g(f(a))) = (h \circ g)(f(a)) = ((h \circ g) \circ f)(a)$$

The proof may be illustrated by this picture:



Each of the two triangles ABC, BDC 'commutes': that is, the result is the same whether one follows f followed by g or $g \circ f$, and whether one does g followed by h or $h \circ g$. The parallelogram ABCD therefore 'commutes', which means that the result holds. □

PROPOSITION 4.16
let $f : A \to B$ and $g : B \to C$ be arbitrary functions. If f, g are bijections, then so is $g \circ f$.

Proof It is enough to show that

1. if f, g are onto then so is $g \circ f$;
2. if f, g are one-to-one then so is $g \circ f$.

To prove the onto result, assume f and g are onto and let c be an arbitrary element of C. Since g is onto, we can find an element $b \in B$ such that

$g(b) = c$. Since f is onto, we can also find an element $a \in A$ such that $f(a) = b$. But then $g \circ f(a) = g(f(a)) = g(b) = c$, and hence $g \circ f$ is onto.

To prove the one-to-one result, assume f and g are one-to-one. Let a_1, a_2 be arbitrary elements of A, and suppose $g \circ f(a_1) = g \circ f(a_2)$. Then $g(f(a_1)) = g(f(a_2))$ by the definition of $g \circ f$. Since g is one-to-one, it follows that $f(a_1) = f(a_2)$. Since f is also one-to-one, it follows that $a_1 = a_2$, and hence $g \circ f$ is one-to-one. □

DEFINITION 4.17 (IDENTITY FUNCTION)
Let A be a set. Define the *identity* function on A, denoted $\mathsf{id}_A : A \to A$, by $\mathsf{id}_A(a) = a$ for all $a \in A$. In Haskell, we would declare the function

```
id :: A -> A
id x = x
```

We shall now define the inverse function. We cannot just define it using the definition of inverse relations, as we do not always get a function this way. For example, the inverse relation of the function $f : \{1, 2\} \to \{1, 2\}$ defined by $f(1) = f(2) = 1$ is not a function. However, we shall see that when inverse functions exist they correspond to the inverse relation.

DEFINITION 4.18 (INVERSE FUNCTION)
let $f : A \to B$ be an arbitrary function. The function $g : B \to A$ is an *inverse* of f if and only if

$$\begin{array}{rrcl} \text{for all } a \in A, & g(f(a)) & = & a \\ \text{for all } b \in B, & f(g(b)) & = & b \end{array}$$

Another way of stating the same property is that $g \circ f = \mathsf{id}_A$ and $f \circ g = \mathsf{id}_B$.

Example Let $A = \{a, b, c\}$, $B = \{1, 2, 3\}$, $f = \{(a, 1), (b, 3), (c, 2)\}$ and $g = \{(1, a), (2, c), (3, b)\}$. Then g is an inverse of f.

PROPOSITION 4.19
Let $f : A \to B$ be a bijection, and define $f^{-1} : B \to A$ by

$$f^{-1}(b) = a \text{ whenever } f(a) = b$$

In this case, the relation f^{-1} is a well-defined function, and is an inverse of f (in fact, *the* inverse in view of the next proposition).

Proof Let $b \in B$ be arbitrary. Since f is onto, there is an a such that $f(a) = b$. Since f is one-to-one, this a is unique. This means that f^{-1} is a function. By definition, it satisfies the conditions for being an inverse of f. □

PROPOSITION 4.20
Let $f : A \to B$. If f has an inverse g, then f must be a bijection and the inverse is unique (and is the f^{-1} given in proposition 4.19).

Proof Let g be the inverse of f. To show that f is onto, let b be an arbitrary element of B. Since $f(g(b)) = b$, it follows that b must be in the image of f. To show that f is one-to-one, let a_1, a_2 be arbitrary elements of A. Suppose $f(a_1) = f(a_2)$, which implies that $g(f(a_1)) = g(f(a_2))$. Since $g \circ f = \mathsf{id}_A$, it follows that $a_1 = a_2$. To show that the inverse is unique, suppose that g, g' are both inverses of f. We will show that $g = g'$. Let b be an arbitrary element of B. Then $f(g(b)) = f(g'(b))$ since g, g' are inverses. Hence $g(b) = g'(b)$ since f is one-to-one. □

In view of the proceeding proposition, one way of showing that a function is a bijection is to show that it has an inverse. Furthermore, if f is a bijection with inverse f^{-1}, then f^{-1} has an inverse, namely f, and so f^{-1} is also a bijection.

EXAMPLE 4.21
The function $f : \mathcal{N} \to \mathcal{N}$ defined by

$$\begin{aligned} f(x) &= x + 1 \quad x \text{ odd} \\ &= x - 1 \quad x \text{ even} \end{aligned}$$

It is easy to check that $(f \circ f)(x) = x$, considering the cases when x is odd and even separately. Therefore f is its own inverse, and we can deduce that it is a bijection.

4.6 Cardinality of Sets

We are finally in a position to compare the size of *infinite* sets, using a natural relation between sets defined using bijective functions.

DEFINITION 4.22
For any sets A, B, define $A \sim B$ if and only if there is bijection from A to B.

PROPOSITION 4.23
The relation $\sim$ is reflexive, symmetric and transitive.

Proof The relation $\sim$ is reflexive , since $\mathsf{id}_A : A \to A$ is clearly a bijection. To show that it is symmetric, observe that by definition $A \sim B$ implies that there is a bijection $f : A \to B$. By proposition 4.19, it follows that f has an inverse f^{-1} which is also a bijection. Hence $B \sim A$. The fact that the relation $\sim$ is transitive follows from proposition 4.16. □

EXAMPLE 4.24
Let A, B, C be arbitrary sets, and consider the products $(A \times B) \times C$ and $A \times (B \times C)$. In section 2.2.5, we observed that these sets are not in general equal. There is however a natural bijection $f : (A \times B) \times C \to A \times (B \times C)$ given by:

$$f : ((a, b), c) \mapsto (a, (b, c))$$

Define the function $g : A \times (B \times C) \to (A \times B) \times C$ in a similar fashion:

$$g : (a, (b, c)) \mapsto ((a, b), c)$$

It is not difficult to show that $g \circ f = \mathsf{id}_{A\times(B\times C)}$ and $f \circ g = \mathsf{id}_{(A\times B)\times C}$, and hence by proposition 4.20 that f is a bijection.

EXAMPLE 4.25
Consider the set Even of even natural numbers. There is a bijection between Even and $\mathcal{N}$ given by $f(n) = 2n$. Notice that there not all function from Even to $\mathcal{N}$ are bijections: for example, the function $g : \mathsf{Even} \to \mathcal{N}$ given by $g(n) = n$ is one-to-one but not onto. To show that $\mathsf{Even} \sim \mathcal{N}$ it is enough to show the *existence* of such a bijection.

Recall that the cardinality of a *finite set* is the number of elements in that set. Consider a finite set A with cardinality n. Then there is a 'counting' bijection

$$c_A : \{1, 2, \ldots, n\} \to A$$

This function should be familiar, in that we often enumerate the elements in A by $\{a_1, \ldots, a_n\}$. Now let A and B be two finite sets. If A and B have the same number of elements, the we can define a bijection $f : A \to B$ by

$$f(a) = (c_B \circ c_A^{-1})(a)$$

Thus, two finite sets have the same number of elements if and only if there is a bijection between then. We extend this observation to compare the size of *infinite* sets.

DEFINITION 4.26 (CARDINALITY)
Given two arbitrary sets A and B, then A has the same *cardinality* as B, written $|A| = |B|$, if and only if $A \sim B$.

We explore examples of infinite sets in two stages. We first look at those sets which have the same cardinality as the natural numbers, since such sets have nice properties. We then briefly explore examples of infinite sets which

are 'bigger' than the set of natural numbers. The set of natural numbers is one of the simplest infinite sets. We can build it up by stages:

$$\begin{array}{l} 0 \\ 0, 1 \\ 0, 1, 2 \\ \vdots \end{array}$$

in such a way that any number n will appear at some stage. Infinite sets which can be built up in finite portions by stages are particularly nice for computing. We therefore distinguish those set which are either finite or which have a bijection to $\mathcal{N}$.

DEFINITION 4.27 (COUNTABLE)
For any set A, A is *countable* if and only if A is finite or $A \sim \mathcal{N}$. The elements of a countable set A can be listed as a finite or infinite sequence of distinct terms: $A = \{a_1, a_2, a_3, \ldots\}$.

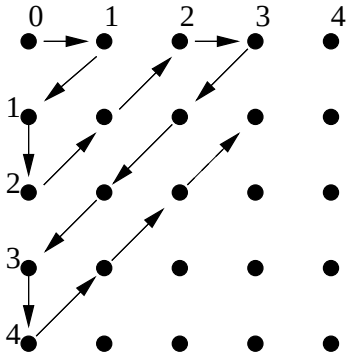
EXAMPLE 4.28
The integers $\mathcal{Z}$ are countable, since they can be listed as:

$$0, 1, -1, 2, -2, 3, -3, \ldots$$

This 'counting' function can be defined formally by the bijection $g : \mathcal{Z} \to \mathcal{N}$ defined by

$$\begin{array}{rcl} g(x) & = & 2x, \quad x \geq 0 \\ & = & -1 - 2x, \quad x < 0 \end{array}$$

EXAMPLE 4.29
The set of integers $\mathcal{Z}$ is like two copies of the natural numbers $\mathcal{N}$. We can even count $\mathcal{N}^2$, which is like $\mathcal{N}$ copies of $\mathcal{N}$, as illustrated by the following diagram:



Comment The rational numbers are also countable.

In contrast, Cantor showed that there are *uncountable* sets: that is, infinite sets that are too large to be countable. An important example is the set of reals $\mathcal{R}$. We cannot build up the reals in infinite stages, and this means that we cannot manipulate reals in the way we can natural numbers. Instead, we have to use approximations, such as the floating point decimals (given by the type `Float` in Haskell). Another example is the power set $\mathcal{P}(\mathcal{N})$. For further information about infinite sets and countability see Truss, section 2.4.

5 Orderings

Orderings are special relations which characterise when one object is 'better' than another. Orderings on sets of numbers such as $\mathcal{N}$, $\mathcal{Z}$ and $\mathcal{R}$ are familiar: the ordering $<$ describes the 'less than' ordering, and $\leq$ the 'less than or equal' ordering. We can also have orderings on other sets. For instance, suppose that we have a set of programs and we wish to distinguish which are cheaper, or run faster, or are more accurate. These sort of orderings are used in Discrete Mathematics 2 to compare the efficiency of algorithms. Here we give the formal definitions and properties of some orderings.

DEFINITION 5.1 (ORDERINGS)

1. Let R be a binary relation on a set A. Then R is a *pre-order* if and only if R is reflexive and transitive.

2. Let R be a binary relation on a set A. Then R is *anti-symmetric* if and only if $\forall a, b \in A.\, (x\, R\, y \wedge y\, R\, x \Rightarrow x = y)$. The relation R is a *partial order* if and only if R is reflexive, transitive and anti-symmetric. Partial orders are often denoted by $\leq$. We write $(A, \leq)$ to denote a partial order $\leq$ on A.

3. Let R be a binary relation on a set A. Then R is *irreflexive* if and only if $\forall a \in A. \neg(a\, R\, a)$. Let R be a binary relation on a set A. Then R is a *strict partial order* if and only if R is irreflexive and transitive. Strict partial orders are often denoted by $<$.

4. A partial order R on A is a *total order* iff $\forall a, b \in A.\, (a\, R\, b \vee b\, R\, a)$. Total orders are also sometimes called *linear orders*.

In the definition of partial order, notice that anti-symmetric is not the opposite of symmetric, since a relation can be both symmetric and anti-symmetric: for example, the identity relation or the empty relation.

EXAMPLE 5.2

1. The numerical orders $\leq$ on $\mathcal{N}$, $\mathcal{Z}$ and $\mathcal{R}$ are total orders. The orders $<$ are strict partial orders.

2. Division on $\mathcal{N} \setminus \{0\}$ is a partial order: $\forall n, m \in \mathcal{N}.\ n \leq m$ iff n divides m.

3. For any set A, the power set of A ordered by subset inclusion is a partial order.

4. Suppose $(A, \leq_A)$ is a partial order and $B \subseteq A$. Then $(B, \leq_B)$ is a partial order, where $\leq_B$ denotes the restriction of $\leq_A$ to the set B.

5. Define a relation on formulae by: $A \leq B$ if and only if $\vdash A \rightarrow B$. Then $\leq$ is a pre-order. For example, $\mathsf{false} \leq A \leq \mathsf{true}$ and $A \leq A \vee B$.

6. For any two partially ordered sets $(A, \leq_A)$ and $(B, \leq_B)$, there are two important orders on the product set $A \times B$:

   - product order: $(a_1, b_1) \leq_P (a_2, b_2)$ iff $(a_1 \leq_A a_2) \wedge (b_1 \leq_B b_2)$
   - lexicographic order: $(a_1, b_1) \leq_L (a_2, b_2)$ iff $(a_1 <_A a_2) \vee (a_1 = a_2 \wedge b_1 \leq_B b_2)$.

   If $(A, \leq)$ and $(B, \leq)$ are both total orders, then the lexicographic order on $A \times B$ will be total. By contrast, the product order will in general only be partial. For any partially ordered sets $(A, \leq)$ and $(B, \leq)$, the product order is contained in the lexicographic order.

7. (For interest, gives ordering for words in a dictionary) For any totally ordered (finite) alphabet A, the sets $A^* = \{\epsilon\} \cup A \cup A^2 \cup A^3 \cup \ldots$ is the set if all strings made from that alphabet, with ϵ denoting the empty string. The *full lexicographic order* $\leq_F$ on A^* is defined as follows. Given two words $u, v \in A^*$, if $u = \epsilon$ then $u \leq_F v$ and if $v = \epsilon$ then $v \leq_F u$. Otherwise, both u and v are non-empty so we can write $u = u_1 x$ and $v = v_1 y$ where u_1 and v_1 are the first letters of u and v respectively. Now $u \leq_F v \Leftrightarrow (u = \epsilon) \vee (u_1 <_A v_1) \vee (u_1 = v_1 \wedge x \leq_F y)$.

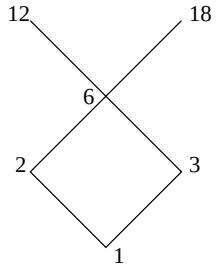
5.1 Hasse Diagrams

Since partial orders are special binary relations on a set A, we can represent them by directed graphs. However, these graphs get rather cluttered if every arrow is drawn. We therefore introduce *Hasse diagrams*, which provide a compact way of representing the partial order. First we require some definitions.

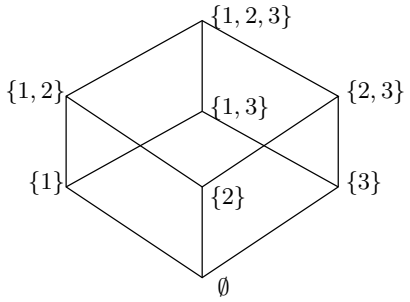
DEFINITION 5.3
If R is a partial order on a set A and $a\,R\,b$ for $a \neq b$, we call a a *predecessor* of b, and similarly b a *successor* of a. If a is a predecessor of b and there is no c with $a\,R\,c$ and $c\,R\,b$, then a is the *immediate predecessor* of b.

Hasse diagrams are like directed graphs, except that they just record the immediate predecessors; the other pairs in the partial order can be inferred. Also the direction of the lines is usually omitted, with the convention that all lines are directed up the page. We give two examples of Hasse diagrams.

EXAMPLE 5.4
For example, the Hasse diagram for the relation 'is a divisor of' for the set $\{1, 2, 3, 6, 12, 18\}$ is



EXAMPLE 5.5
The Hasse diagram for the binary relation $\subseteq$ on $\mathcal{P}(\{1, 2, 3\})$ is



5.2 Analysing Partial Orders

The shape of the partial orders $(\mathcal{N}, \leq)$ and $(\mathcal{Z}, \leq)$ are different from each other. The number 0 is the smallest element with respect to the natural numbers, but not with respect to the integers.

DEFINITION 5.6 (ANALYSING PARTIAL ORDERS)
Let $(A, \leq)$ be a partial order.

1. An element $a \in A$ is *minimal* iff $\forall b \in A.\, (b \leq a \Rightarrow b = a)$.
2. An element $a \in A$ is *least* iff $\forall b \in A.\, a \leq b$.
3. An element $a \in A$ is *maximal* iff $\forall b \in A.\, (a \leq b \Rightarrow a = b)$.
4. An element $a \in A$ is *greatest* iff $\forall b \in A.\, b \leq a$.

In example 5.4, the least (and minimal) element is 1, the maximal elements are 12 and 18, and there is no greatest element. In example 5.5, the least (and minimal) element is $\emptyset$, and the greatest (and maximal) element is $\{1, 2, 3\}$. With the usual partial order $(\mathcal{N}, \leq)$, the least element is 0, and there is no maximal element.

PROPOSITION 5.7
Let $(A, \leq)$ be a partial order.

1. If a is a least element, then a is a minimal element.
2. If a is a least element, then it is unique.
3. If A is finite and non-empty, then $(A, \leq)$ must have a minimal element.
4. If $(A, \leq)$ is a total order, where A is finite and non-empty, then it has a least element.

Proof To prove part 1, suppose that $a \in A$ is least and assume for contradiction that $b < a$ for some $b \in A$ such that $b \neq a$. But $a \leq b$ by definition of a being least. This contradicts anti-symmetry. To prove part 2, suppose that a and b are both least elements. By definition of least element, we have $a \leq b$ and $b \leq a$. By anti-symmetry, it follows that $a = b$.

To prove part 3, pick any $a_0 \in A$. If a_0 is not minimal we can pick $a_1 < a_0$. If a_1 is not minimal, we can pick $a_2 < a_1$. In this way, we get a decreasing chain $a_0 > a_1 > a_2 > \ldots$. All the elements of the chain must be different, by construction. Since A is a finite set, we must find a minimal element at some point. [Notice that in this proof we not only show the existence of minimal elements, but also how to find one.] Part 4 is left as an exercise. □

5.3 From Partial to Total Orders

Given a finite partial order, we can extend it to a total order. For example, suppose we have a set of tasks T to perform. We wish to decide in what order to perform them. We are not totally free to choose, because some tasks have to be finished before others can be started. We can express this pre-requisite structure by a partial order $<$ on T. We want to find a total order $<'$ on T which respects $<$ in the sense that if $t < u$ then $t <' u$.

As a more concrete example, consider the partial order $\subseteq$ on $\mathcal{P}(\{1,2,3\})$ given in example 5.5. It is partial because, for example, the sets $\{1\}$ and $\{3\}$ are not contained in each other. A total order $\subseteq_T$ which extends this partial order is given by the sequence

$$\emptyset, \{3\}, \{2\}, \{1\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}$$

We are forced to have $\{1\} \subseteq_T \{1,2\}$ since we wish to respect the partial order, but we have chosen to have $\{3\} \subseteq_T \{1\}$. This process of going from a partial to a total order is called *topological sorting*, and we can define a simple algorithm for topological sorting based on minimal elements.

PROPOSITION 5.8 (TOPOLOGICAL SORTING)
Let $(A, \leq)$ denote a finite partial order. We can construct a *total* order $\leq_T$ on A such that $\forall a, b \in A.(a \leq b \Rightarrow a \leq_T b)$.

Proof First *choose* a minimal element $a_1 \in A$. Such an element exists since A is finite. Note that $(A \backslash \{a_1\}, \leq)$ is also a po. If it is non-empty, *choose* a minimal element $a_2 \in A \backslash \{a_1\}$. Continue this process, until there are no more elements left. [In fact there is a slight subtlety. At each step, there may be more than one minimal element. These cannot be compared with each other, so it does not matter what order they have in the total order. Instead of putting just one of them into the total order, we could include all of them in some arbitrary order before going on to the next step and finding the minimal elements in the remainder of A.] Since A is finite, this process must terminate. The total order is given by the sequence $a_1, a_2, a_3, \ldots$. □

5.4 Well-founded Partial Orders

Well-founded partial orders are extremely useful. For example, consider the function $\mathsf{Ack} : \mathcal{N} \times \mathcal{N} \to \mathcal{N}$ defined by

$$\begin{aligned} \mathsf{Ack}(0, y) &= y + 1 \\ \mathsf{Ack}(x+1, 0) &= \mathsf{Ack}(x, 1) \\ \mathsf{Ack}(x+1, y+1) &= \mathsf{Ack}(x, \mathsf{Ack}(x+1, y)) \end{aligned}$$

This program is a well-known example in computer science, since the function computed by this program grows extremely rapidly. We wish to prove that this program always terminates, and therefore defines a total function. Counting down from x is not good enough, since the third equation does not decrease $x + 1$, because of the embedded $\mathsf{Ack}(x + 1, y)$. We will devise a different way of counting down, by defining a well-founded partial order with the property that it always decreases to a terminating state.

DEFINITION 5.9 (WELL-FOUNDED PARTIAL ORDERS)
A *partial order* $(A, \leq)$ is *well-founded* if and only if it has no infinite decreasing chain of elements: that is, for *every* infinite sequence $a_1, a_2, a_3, \ldots$ of elements in A with $a_1 \geq a_2 \geq a_3 \geq \ldots$, there exists $m \in \mathcal{N}$ such that $a_n = a_m$ for every $n \geq m$.

For example, the conventional numerical order $\leq$ on $\mathcal{N}$ is a well-founded partial order. This is *not* the case for $\leq$ on $\mathcal{Z}$, which can decrease for ever.

PROPOSITION 5.10
If two partial orders $(A, \leq)$ and $(B, \leq)$ are well-founded, then the lexicographical order on $A \times B$ (see example 5.2) is also well-founded.

Proof Suppose $(a_1, b_1) \geq_L (a_2, b_2) \geq_L (a_3, b_3) \geq_L \ldots$. Then $a_1 \geq_A a_2 \geq_A a_3 \geq_A \ldots$ by the definition of lexicographic order, so the sequence is must ultimately consists of the same element, since $(A, \leq_A)$ is well-founded. Therefore, there exists $m \in \mathcal{N}$ such that $a_n = a_m$ for every $n \geq m$. Now by the definition of lexicographic order, we have $b_m \geq_B b_{m+1} \geq_B b_{m+3} \geq_B \ldots$, and this sequence must also ultimately end up being constant because $(B \leq_B)$ is well-founded. Thus, the original sequence is ultimately constant. □
This result implies that the product order on $A \times B$ is well-founded, since the product order is contained in the lexicographical order (see example 5.2).

Let us return to the Ack function, and consider the strict lexicographical order on $\mathcal{N} \times \mathcal{N}$ by

$$(x, y) < (x', y') \text{ if and only if } x < x' \text{ or } (x = x' \text{ and } y < y')$$

Notice that

$$\begin{aligned} (x + 1, 0) &> (x, 1) \\ (x + 1, y + 1) &> (x, \mathsf{Ack}(x + 1, y)) \\ (x + 1, y + 1) &> (x + 1, y) \end{aligned}$$

and so evaluating the Ack function takes us down the order. Moreover the order is well-founded, using proposition 5.10 and the fact that $(\mathcal{N}, <)$ is

well-founded. Even though a member such as $(4, 3)$ has infinitely many other elements below it (for example, $(3, y)$ for every y), any decreasing chain must be finite. Hence, the Ack program always gives an answer.