

Chapter 2

Mathematical Background

We start this chapter with a summary of basic concepts and notations for sets and first-order logic formulas, which will be used in the rest of the book. Our aim is to try to make the book self contained; readers who are familiar with set theory and first-order logic can skip the first two sections of this chapter and go directly to Sect. 2.3, where we define transition systems. Section 2.4 introduces induction as a technique to define sets and to prove their properties.

2.1 Sets and Relations

We will denote the *empty set*, that is, a set with no elements, by \emptyset or $\{\}$. Non-empty sets will be given extensionally when possible, by enumerating their elements between curly brackets. For instance, the set of Boolean values can be given extensionally as follows.

$$Booleans = \{True, False\}$$

To indicate that an element is in a set we use the symbol \in , as in $True \in Booleans$. The symbol \in denotes membership; its negation is written \notin , as in $0 \notin Booleans$.

When sets are infinite, it is not possible to enumerate all the elements so we will define them intensionally, by characterising their elements. We do this by giving a property that all the elements in the set satisfy. For example, even natural numbers are characterised by the fact that the remainder of the division by 2 is 0, that is, a natural number n is even if $n \bmod 2 = 0$. Thus, if \mathcal{N} is the set of natural numbers we can define the set of even natural numbers intensionally as follows.

$$Even = \{n \in \mathcal{N} \mid n \bmod 2 = 0\}$$

This kind of definition is called definition by comprehension.

Another way of defining sets intensionally is by using induction; we will discuss induction in Sect. 2.4.

Two essential notions in set theory are inclusion and equality of sets.

Definition 2.1 (Set inclusion, equality)

A set A is *included* in the set B , written $A \subseteq B$, if all the elements in A are also in B . In this case, we say that A is a *subset* of B .

Using inclusion, we can define equality of sets: Two sets are *equal* if they have exactly the same elements, that is, $A = B$ if and only if $A \subseteq B$ and $B \subseteq A$.

Therefore, to check that two sets are equal we need to check that they contain the same elements. For example, the sets $\{True, False\}$ and $\{False, True, True\}$ are equal. This is an instance of a more general property: the order in which we write the elements of a set is irrelevant, and repeating elements is not very elegant but does not change the set either.

The standard operations on sets are:

- *Union*, written \cup , which, given two sets A and B builds a new set containing all the elements in A and all the elements in B :

$$A \cup B = \{e \mid e \in A \text{ or } e \in B\}$$

Thus, the set $A \cup B$ is empty only if A and B are both empty.

Union is an associative and commutative operator, that is:

$$\begin{aligned} (A \cup B) \cup C &= A \cup (B \cup C) \\ A \cup B &= B \cup A \end{aligned}$$

Since union is associative, we can write $A \cup B \cup C$ without brackets (there is no confusion since any choice of brackets produces the same result).

- *Intersection*, written \cap , which, given two sets A and B builds a new set containing the elements that are in both sets:

$$A \cap B = \{e \mid e \in A \text{ and } e \in B\}$$

Intersection is also an associative and commutative operator:

$$\begin{aligned} (A \cap B) \cap C &= A \cap (B \cap C) \\ A \cap B &= B \cap A \end{aligned}$$

The intersection of two sets is empty when they have no elements in common; in particular, it is empty if either of the sets is empty. Since intersection is associative, we can write $A \cap B \cap C$ without brackets.

- *Set difference* (or just difference), written $-$, which, given two sets A and B builds a new set as follows:

$$A - B = \{e \in A \mid e \notin B\}$$

Thus, the difference set $A - B$ contains all the elements that are in A but not in B . Another way of defining this set is as a *relative complement*: it is the complement

of B with respect to A . This is sometimes written as \overline{B} (read complement of the set B), when the set A is obvious from the context.

Set difference is neither commutative nor associative.

- *Cartesian product* (or just product), written \times , which, given two sets A and B builds the set containing all the pairs where the first component is in A and the second in B :

$$A \times B = \{(a, b) \mid a \in A \text{ and } b \in B\}$$

Since we are dealing with ordered pairs here, product is neither commutative nor associative on sets.

This operation can be generalised to products of a finite number of sets. Thus, $A_1 \times \dots \times A_n$ is a set of tuples (a_1, \dots, a_n) where each of the elements a_i ($1 \leq i \leq n$) is in the set A_i .

Note that we have defined the sets resulting from an operation of union, intersection, difference or product by comprehension: we have specified the properties that characterise the elements of the resulting sets. We have relied on an intuitive understanding of the connectives “or”, “and” and “not”; we will define them precisely in the next section.

Now, we will use products of sets to define relations.

Definition 2.2 (Relation)

A binary *relation* R between sets A and B is a subset of the Cartesian product $A \times B$:

$$R \subseteq A \times B$$

In other words, a binary relation R is a subset of the pairs where the first element is in A and the second in B . We will use the notation $R(a, b)$ to indicate that the pair (a, b) is in the relation R .

If the sets A and B coincide, we will simply say that R is a relation on A .

Binary relations will be used in the next section to define transition systems associated to programs, where a transition will relate two states in the system. In this context, we will consider two properties of relations:

Definition 2.3 A binary relation $R \subseteq A \times A$ is *reflexive* if it includes all the pairs (a, a) such that $a \in A$. It is *transitive* if each time we have $R(a, b)$ and $R(b, c)$ we also have $R(a, c)$.

If a relation on A is not reflexive, we can make it reflexive by adding the missing pairs (a, a) , and similarly, if it is not transitive, we can make it transitive by adding the pair (a, c) whenever there exists b in A such that $R(a, b)$ and $R(b, c)$.

Definition 2.4 The *reflexive-transitive closure* of the relation $R \subseteq A \times A$ is the least reflexive and transitive relation on A that contains R . It is denoted by R^* .

The reflexive-transitive closure of the relation R is obtained simply by extending R with all the pairs needed to satisfy reflexivity and transitivity.

If we represent R as a graph (i.e., as a set of nodes and edges, where the nodes are the elements of A and there is an edge from a to b if and only if (a, b) is in R),

then the reflexive-transitive closure of R is the relation representing reachability: if (a, b) is in R^* then there is a path from a to b in the graph. This idea will be useful later: we will use transition relations to specify the behaviour of programs, and the reflexive-transitive closure will represent sequences of computations.

Another useful notion is the closure of a set under an operator. In general, a set is said to be closed under an operation if the result of the operation on elements of the set is also a member of the set. The notion of closure can be stated using relations.

Definition 2.5 Let R be a binary relation on A . We will say that a subset A' of A is closed under the relation R if R relates elements of A' with elements of A' .

Again, this notion will be useful later, in particular to provide an inductive definition of the transition relation associated to a set of programs.

Some relations are *functions*. We can characterise functions from A to B as relations that associate at most one element in B with each element in A : for each $a \in A$, there is at most one $b \in B$ such that a and b are related.

Definition 2.6 A relation $R \subseteq A \times B$ is a function if $R(a, b)$ and $R(a, c)$ implies $b = c$, for every $a \in A$.

To emphasise the fact that a relation R is a function from A to B , we will write $R: A \rightarrow B$, and denote by $R(a)$ the unique element in B related with $a \in A$.

For example, the relation that associates each natural number n with the number $n + 1$ is a function, usually called the *successor* function S :

$$S: \mathcal{N} \rightarrow \mathcal{N} = \{(n, n + 1) \mid n \in \mathcal{N}\}$$

Another way of defining S is by writing $S(n) = n + 1$.

The notions of domain and image of a function can be formally defined:

If R is a function from A to B , its *domain* is the subset of A consisting of all the elements for which there exists an element b in B such that $(a, b) \in R$. In other words, the domain of the function R consists of all the elements a in A such that $R(a)$ is defined.

If R is a function from A to B , its *image* is the subset of B consisting of all the elements for which there exists an element a in A such that $(a, b) \in R$. In other words, the image of the function R consists of all the elements b in B for which there is some a in A such that $b = R(a)$. The element $R(a)$ is the *image of a* under R .

So far, we have considered only binary relations, however, the ideas can be extended to n -ary relations. An n -ary relation R on the sets A_1, \dots, A_n is a set of tuples (a_1, \dots, a_n) where each of the elements a_i is in the set A_i . Thus, $R \subseteq A_1 \times \dots \times A_n$. We will use the notation $R(a_1, \dots, a_n)$ to indicate that the tuple (a_1, \dots, a_n) is in the relation R . Thus, we can use a relation to characterise a set of tuples with a relevant property. In other words, we can see a relation as a predicate (indeed, relations are used to give a meaning to predicates in first-order logic).

2.2 Logic Formulas

In this section, we define the syntax of formulas in first-order logic and recall briefly their semantics. First we need to define *terms*.

Definition 2.7 (Terms)

Given an infinite but denumerable set \mathcal{X} of symbols x, y, z, x_1, x_2 , etc., which we call *variables*, and an infinite but denumerable set \mathcal{F} of symbols f, g, h, f_1, f_2 , etc., which we call *function symbols*, each with an associated arity (a natural number), the set of terms built out of \mathcal{X} and \mathcal{F} is generated by the grammar:

$$T \rightarrow x \mid f(T, \dots, T)$$

where x is an element of \mathcal{X} , f is in \mathcal{F} and if the arity of f is n then any occurrence of f in a term must be applied to exactly n terms t_1, \dots, t_n .

The grammar given above describes the abstract syntax of terms. In other words, terms are either variables or are built using a function symbol and a number of previously built terms corresponding to the arity of the function symbol. We say that t_1, \dots, t_n are the arguments of f in the term $f(t_1, \dots, t_n)$. Thus, an n -ary function symbol must always have exactly n arguments. Note that n could be zero, in this case the function symbol is called a *constant*.

For example, we can define a language of terms representing natural numbers using just two function symbols: *zero*, of arity 0, and *suc* of arity 1. In this case, we do not need variables. The number 0 will be represented by the term *zero()*. To improve readability, when function symbols have arity 0 we will omit the brackets. Thus, to represent the number 0 we write simply *zero*; similarly, the term *suc(suc(zero))* represents the number 2. We will sometimes write *suc(n)* as $n + 1$.

Formulas in first-order logic are built using a set of terms, a set of *predicate symbols*, each with a given arity, and additionally a set of *connectives* and *quantifiers* (see below). Terms and predicate symbols are used to build *atomic formulas*, also called *atoms*, of the form $P(t_1, \dots, t_n)$, where P is a predicate of arity n and t_1, \dots, t_n are terms. We will distinguish two 0-ary predicates, T and F , which will denote true and false values, respectively.

For example, using the binary predicate $>$, we can write the atomic formula $>(suc(zero), zero)$. We will usually write this predicate using an infix notation, that is, we write *suc(zero) > zero*.

Atomic formulas may also have variables, for example, we could write *suc(x) > zero*. In this case, we will say that the variable x is *free* in the formula.

To build more complex formulas, in this book we will use the binary connectives \wedge (read “and”), \vee (read “or”), \Rightarrow (read “implies”), and the unary connective \neg (read “not”). So, if ϕ and ψ are logic formulas, then $\phi \wedge \psi$, $\phi \vee \psi$, $\phi \Rightarrow \psi$ and $\neg\phi$, are also formulas. In addition, first-order logic formulas may have *quantifiers* \forall and \exists (read “for all” and “there exists”), denoting universal and existential quantification, respectively: If ϕ is a formula and x is a variable, then $\forall x.\phi$ and $\exists x.\phi$ are formulas.

In a formula of the form $\forall x.\phi$ or $\exists x.\phi$, the variable x is said to be *quantified* or *bound*.

Formally, the language of first-order formulas can be defined using a grammar; see the Exercises at the end of this chapter. A precise definition of the sets of free and bound variables in a formula is also left as an exercise. A formula without free variables is said to be *closed*.

Once we have defined the syntax of the formulas, we can try to associate a meaning (a truth value) to each formula. However, if the formula has free variables, its truth value depends on the values of the variables, so the meaning of the formula can only be given in relation to a *valuation* that assigns values to the variables. In addition, to establish the truth value of the formula we need to know how to interpret the function symbols and predicates.

We describe now how to map formulas to truth values.

We start by fixing a domain U of interpretation for the variables; this is sometimes called a *universe*. Now, assume each function symbol of arity n is interpreted by an n -ary function on U (in the particular case of $n = 0$, constants are interpreted by elements of U) and each predicate of arity n is interpreted by an n -ary relation on U . We will denote by $[f]$ the function associated to the function symbol f , and by $[P]$ the relation associated to the predicate symbol P . If the formula is closed, we do not need a valuation: the truth value of the formula depends only on the chosen universe, the meaning of the function symbols and the meaning of predicate symbols. Otherwise, assume we are also given a valuation σ mapping variables to elements of U .

Using the valuation σ and the interpretations of function symbols we can give a meaning to terms. Then, using the interpretations of terms and predicate symbols, we can give denotations for formulas.

Definition 2.8 (Denotation of terms)

The interpretation, or *denotation*, of a term t in U with respect to a valuation σ and interpretations for function symbols, written $[t]$, is defined as follows.

- The denotation of a variable x is $\sigma(x)$:
 $[x] = \sigma(x)$
- The denotation of $f(t_1, \dots, t_n)$ is the image of $([t_1], \dots, [t_n])$ under $[f]$:
 $[f(t_1, \dots, t_n)] = [f]([t_1], \dots, [t_n])$

Once we know the meaning of terms, we can specify the truth values of formulas.

Definition 2.9 (Denotation of formulas)

A formula $P(t_1, \dots, t_n)$ is true if $([t_1], \dots, [t_n]) \in [P]$, that is, if $[P]([t_1], \dots, [t_n])$; otherwise it is false. For the distinguished symbols T and F , we define $[T]$ to be true and $[F]$ to be false. Finally, the connectives have the following meanings:

- \wedge denotes conjunction, that is, $\phi \wedge \psi$ is true if both ϕ and ψ are true;
- \vee denotes disjunction, that is, $\phi \vee \psi$ is true if at least one of ϕ or ψ are true;

- \Rightarrow denotes implication, $\phi \Rightarrow \psi$ is true if each time ϕ is true, ψ is also true;
- \neg denotes negation: $\neg\phi$ is true if ϕ is false, and it is false if ϕ is true.

In addition, quantified formulas are interpreted as follows: the formula $\forall x.\phi$ is true if ϕ is true for every possible value of x , and the formula $\exists x.\phi$ is true if there is at least one value of x that satisfies ϕ (that is, it makes it true).

According to the previous definitions, under an interpretation each term has a value in U , and each formula is either true or false.

Example 2.10

Let us assume we interpret the variable x on the natural numbers, so our universe is \mathcal{N} .

To assign a meaning to the formula $suc(x) > zero$ we need a valuation, which will give us a value for x , let us say 0, and we need interpretations for the function symbols $zero$ and suc and predicate symbol $>$. Let us assume we interpret $zero$ as the number 0, suc as the successor function and $>$ as the relation containing all the pairs of natural numbers where the first element is greater than the second. Then the formula $suc(x) > zero$ with the valuation that assigns 0 to x is true under this interpretation. Actually, this formula is true for any valuation, if we interpret the variable x on the natural numbers.

Therefore, the formula $\forall x.suc(x) > zero$ is true if we interpret the variable x on the natural numbers, assuming $[suc]$ is the successor function, $[zero]$ is the number 0, and $[>]$ is the relation containing all the pairs of natural numbers where the first element is greater than the second.

The formula $zero > suc(x)$ is false for all valuations in \mathcal{N} , but some formulas are true for certain valuations and false for others. For example $suc(zero) > x$ is only true for the valuation that assigns to x the value 0.

We will denote the equality predicate by $=$ as usual, and write it in infix notation. Then, we can write a formula $\exists x.suc(x) = zero$, but this formula is not true if we interpret x on the natural numbers and interpret suc as the successor function, $zero$ as the number 0, and $=$ as the relation containing all the pairs of natural numbers where the first element is equal to the second.

According to our definitions, the formulas $\phi \vee \psi$ and $\psi \vee \phi$ have the same meaning: they are both true if at least one of the formulas ϕ , ψ are true, and false if both are false. We conclude that \vee is commutative. Similarly, \wedge is commutative, and both \vee and \wedge are associative.

Note that in the case of quantified formulas, the name of the bound variable is not important, since the formula's truth value (which we are taking as its meaning) does not change if we rename the variable. We will not define the renaming operation formally here, but the intuition is that we can consistently change the name of the variable everywhere in the formula.

The meaning of a quantified formula may of course change if we change the domain of interpretation of the bound variables, that is, if we change the set of values that the quantified variables may take.

To make clear which is the domain of interpretation we have in mind, when we write first-order logic formulas with quantifiers we will write explicitly the domain next to the quantified variable. For instance, we may write $\forall n \in \mathcal{N}.(P(n) \Rightarrow P(suc(n)))$.

2.3 Transition Systems

A transition system is a mathematical device that can be used to model computation.

Definition 2.11 (Transition System)

A *transition system* is specified by

- a set *Config* of *configurations* or *states*;
- a binary relation $\rightarrow \subseteq \text{Config} \times \text{Config}$, called the *transition relation*.

We use the notation $c \rightarrow c'$ (infix) to indicate that c and c' are related by \rightarrow .

The expression $c \rightarrow c'$ denotes a *transition* from c to c' . A transition can be understood simply as a change of state. It will represent a step of computation.

To describe the computation performed by a program, we need to specify sequences of transitions. For this, we can use the reflexive-transitive closure of the relation \rightarrow , denoted by \rightarrow^* . By definition, \rightarrow^* is a binary relation that contains \rightarrow , contains all the pairs $c \rightarrow^* c$ (reflexivity), and all the pairs $c \rightarrow^* c'$ such that $c \rightarrow^* c''$ and $c'' \rightarrow^* c'$ for some c'' (transitivity). Therefore, $c \rightarrow^* c'$ holds if and only if there is a sequence of transitions:

$$c \rightarrow c_1 \rightarrow \dots \rightarrow c_n = c' \quad \text{where } n \geq 0.$$

We will distinguish *initial* and *final* (also called *terminal*) subsets of configurations, written I, T respectively. A final configuration c is characterised by the fact that there is no transition out of c , more precisely: for all c in T , there is no configuration c' such that $c \rightarrow c'$. This can be written in a more compact way as follows.

$$\forall c \in T, c \not\rightarrow .$$

The intuitive idea is that a sequence of transitions from an initial state $i \in I$ to a final state $t \in T$ represents a run of the system.

Definition 2.12 A transition system is *deterministic* if for every configuration c , whenever $c \rightarrow c_1$ and $c \rightarrow c_2$ we have $c_1 = c_2$.

According to the definition, in a deterministic system, at each point of the computation there is at most one possible transition. In other words, the transition relation is a function. In the context of programming languages, this means that if a program is associated to a deterministic transition system, then at each point in the execution the next step of computation is uniquely defined.

In this book, we will be concerned mostly with deterministic programming languages. We will consider non-deterministic systems when giving examples in concurrent languages. Indeed, non-determinism is one of the defining features of concurrent languages.

Historically, transition systems were the first tool used to give the formal semantics of a programming language. In fact, the first formal description of the behaviour of a program was in terms of an abstract machine, which is a particular kind of transition system.

Definition 2.13 (Abstract Machine)

An *abstract machine* for a programming language is a transition system that simulates the execution of programs. An abstract machine specifies how each construct in the language is executed and for this reason can be seen as an interpreter for a programming language.

In Chap. 4 we will give an abstract machine for a simple imperative language.

Abstract machines are very useful for implementing a programming language since they describe the execution of each command, step by step. However, for users of the language they are not always easy to understand: abstract machines deal with implementation details that are not always useful to programmers. For the same reason, abstract machines are not an ideal tool for analysing programs, reasoning about programs or proving properties of programs.

It is possible to specify the operational semantics of the language in a more structured way, abstracting some of the implementation details. The structural approach to operational semantics based on transition systems gives an *inductive* definition of the execution of a program, where the transition relation is defined by induction and each command is described in terms of its components. We will give examples of structural operational semantics for imperative and functional languages, in Chaps. 4 and 6. We end this chapter with an overview of induction principles.

2.4 Induction

Induction is a powerful technique to define sets and prove their properties. An inductive set is usually defined as the closure of a given set under some operations, that is, the inductive set is the least subset of the given set that is closed under the operations.¹

The most well known instances of this technique are the inductive definition of the set of natural numbers, with its associated Principle of Mathematical Induction, and inductive definitions of data structures, with the associated Principle of Structural Induction. We will use structural induction to prove properties of abstract syntax.

¹ More formally, an inductive set is defined as the least fixed point of an operator, which, under certain conditions, is guaranteed to exist.

Another useful version of induction is Rule Induction, which we will use to define transition systems. Its associated Principle of Rule Induction will be used to prove properties of the transition relation.

Mathematical Induction. The set \mathcal{N} of natural numbers is defined inductively, as the *smallest* set that contains the number 0 and is closed under the successor operation. The latter means that if n is a natural number then its successor $n + 1$ is also in the set of natural numbers. Also, since it is the smallest set satisfying this condition, the only elements in \mathcal{N} are 0 and its successors. Instead of writing 0, $0 + 1$, $0 + 1 + 1$, etc., it is traditional to write the elements of the set \mathcal{N} as 0, 1, 2, \dots

The *Principle of Mathematical Induction* says that to prove that a certain property P holds for all the natural numbers, which we write as

$$\forall n \in \mathcal{N} \cdot P(n)$$

to mean “ $P(n)$ is true for every natural number n ”, it is sufficient to prove the property P for 0 and to prove that if P holds for an arbitrary number n , then it holds for $n + 1$. Thus, each proof by mathematical induction has two parts:

- In the first part, called *Basis*, we need to prove $P(0)$, that is, prove that the property holds for the number 0.
- In the second part, called *Induction Step*, we need to prove the property for the number $n + 1$ under the assumption that it is true of the number n . This is written:

$$\forall n \in \mathcal{N} \cdot (P(n) \Rightarrow P(n + 1)).$$

The assumption $P(n)$ in the induction step is called *Induction Hypothesis*.

The reason this is sufficient to deduce that P holds for all natural numbers is that, according to its inductive definition, \mathcal{N} is the *least* set containing 0 and closed under successor. All the natural numbers are generated from 0 using the successor operation, thus, if we can prove the property for 0 and we can prove it is closed under successor, then we have proved that all natural numbers have the property.

The Principle of Mathematical Induction is used frequently in arithmetic, and is a useful technique to reason about programs that manipulate numbers, as the following example shows.

Example 2.14

Suppose that we need to compute the value of an expression that depends on an index i (a natural number). We write exp_i to highlight the fact that the expression exp depends on i . Moreover, suppose that we need to find the total value of expressions with indices in a given range, for example, we need the sum of all the expressions with indices between j and k . We use the notation

$$\sum_{i=j}^k exp_i$$

to represent the sum of $exp_j, exp_{j+1}, \dots, exp_k$, where we assume that j is less than or equal to k . If j is greater than k then we have an empty sum, which is 0 by definition.

For instance, if the expression is just i and the range is 0 to n , then

$$\sum_{i=0}^n i = 0 + 1 + \dots + n$$

Writing a program to compute such a sum is not a difficult task, but if we need to compute a lengthy sum, instead of writing a program that adds the expressions one by one, we can simply compute $n(n+1)/2$. If n is a large number, this is a much better way to obtain the required result. But how can we be sure that this program and the one that computes the lengthy addition produce the same result, for any natural number n ? Program equivalence is in fact a difficult problem (it is undecidable in general). In this particular case, we can easily prove, by induction, that replacing the lengthy addition $0 + 1 + \dots + n$ by $n(n+1)/2$ is indeed a correct optimisation.

The property that we want to prove states that the sum $0 + 1 + \dots + n$ produces the same result as $n(n+1)/2$, for any natural number n . In other words, we want to prove

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

Similarly, if the expression is $2i - 1$ and we need to compute

$$\sum_{i=1}^n (2i - 1) = 1 + 3 + \dots + (2n - 1)$$

we can avoid the lengthy sum and simply output n^2 . To show that this is indeed a valid optimisation, we can simply prove by induction that for any natural number n this sum is equal to n^2 .

According to the Principle of Mathematical Induction, we need to prove the required property first for the base case and then the induction step. We leave the proof of the first property as an exercise (see the list of exercises at the end of the chapter), and show below the proof of the second property, which we can write as

$$\forall n \cdot \left(\sum_{i=1}^n (2i - 1) = n^2 \right)$$

Basis: Here we need to prove the property for $n = 0$, in other words, we need to show

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

The left-hand side is an empty sum, which is 0 by definition, and the right-hand side is also 0, so the basis is trivial in this case.

Induction Step: Assume $\sum_{i=1}^n (2i - 1) = n^2$ (induction hypothesis). We need to prove that $\sum_{i=1}^{n+1} (2i - 1) = (n + 1)^2$. First, notice that

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

Using the induction hypothesis we obtain

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

and the latter is equal to $(n + 1)^2$ as required.

Structural Induction. Programs usually deal with various kinds of data structures, such as strings, lists and trees, in addition to numbers. To prove universal properties of data structures we can use induction on the size of the structure. For example, to show that a certain property P is true for all lists, we can prove that for every natural number n , P holds for all lists of length n . In this way, we translate a property of lists to a property of numbers. However, instead of translating our property, we can adapt the induction principle to work directly on the data structures of interest (specifically, recursively-defined structures such as strings, lists or trees). Let us consider first the case of lists.

Assume we denote an empty list by nil , and a non-empty list by $cons(h, l)$ where h is the head element and l is the tail of the list. For simplicity, let us assume that we are building lists of natural numbers, so h is a natural number. For example, the list containing just the element 0 is written $cons(0, nil)$. The operator $cons$ is a *constructor* for lists, hence its name.

There are only two alternatives: either the list is empty (nil), or it is not empty and in this case it must have a head h and a tail l . Since these are the only possible ways of building lists, we can define the set of lists of numbers as an inductive set, as follows.

Definition 2.15 (Lists of natural numbers)

The set L of lists of natural numbers is the smallest set that contains the empty list nil and is closed under the operator that takes a list l of natural numbers and a natural number h and produces the list $cons(h, l)$. The latter means that if the set L contains a list l , then it also contains $cons(h, l)$ for any natural number h .

For example, the list containing the numbers 1, 3, 5 can be obtained starting with the empty list (nil) and building first $cons(5, nil)$, then $cons(3, cons(5, nil))$, and finally $cons(1, cons(3, cons(5, nil)))$.

According to Definition 2.15:

- nil is a list, and
- if we know that l is a list and h is a natural number, then $cons(h, l)$ is a list.

Moreover, we can state that there are no other cases, since we defined the set L as the *smallest* set that contains nil and is closed under the $cons$ operator.

As a consequence of this observation, we can derive a mechanism to define functions on lists (recursion) and a principle of induction to prove properties on lists (induction on the structure of lists).

To define a function f on lists, and ensure that every element in the inductive set of lists has an image, it is sufficient to define the image of the empty list (i.e., define $f(nil)$) and define the image $f(cons(h, l))$ of a non-empty list $cons(h, l)$, assuming we already know how to compute $f(l)$.

For example, we can define the length of a list by cases as follows.

- $length(nil) = 0$,
- $length(cons(h, l)) = length(l) + 1$.

The principle of induction for lists mimics the definition of the set L .

Definition 2.16 (Principle of induction for lists)

To prove that a property P holds for every list of natural numbers, that is, for every element of the set L we have defined, it is sufficient to prove:

- *Basis*: $P(nil)$. This means that we need to prove that the property holds for the empty list.
- *Induction Step*: $P(l)$ implies $P(cons(h, l))$ for each element h and list l ; this is written:

$$\forall h \in \mathcal{N}, \forall l \in L. (P(l) \Rightarrow P(cons(h, l))).$$

It means that we need to prove the property for the list $cons(h, l)$ under the assumption that it is true of the list l . Here the assumption $P(l)$ is the Induction Hypothesis.

Consider the following definition of the mirror image of a list, which we call its *reverse*.

- $reverse(nil) = nil$,
- $reverse(cons(h, l)) = append(reverse(l), h)$, where the append operation simply adds the element h at the end of the list $reverse(l)$.

Equipped with the principle of induction for lists, we can now prove properties of lists by induction directly, without needing to reword the property as a property of numbers. For instance, we can prove by induction that the reverse of a list is a list of the same length, for any list.

- *Basis*: The reverse of *nil* is also *nil*, so the property holds trivially in this case.
- *Induction Step*: Given a non-empty list $\text{cons}(h, l)$, we remark that, by the induction hypothesis, l and its reverse have the same length n . Since both the original list $\text{cons}(h, l)$ and its reverse have one additional element, we conclude that they both have length $n + 1$.

The induction principle for lists is a particular case of the Structural Induction Principle, which applies to any set built out of a finite set of basic elements e_1, \dots, e_n using a finite set of constructors c_1, \dots, c_k to generate new elements from previously built elements. The set of terms defined in Sect. 2.2 is such a set. The basic elements correspond to variables and 0-ary function symbols, and each function symbol f of arity n greater than 0 can be seen as a term constructor that allows us to generate a new term $f(t_1, \dots, t_n)$ using previously built terms t_1, \dots, t_n .

In general, we can use structural induction to define sets of finite labelled trees. The Structural Induction Principle that we used for lists can be easily adapted to deal with trees, as we will show below. This provides a useful tool to reason on the abstract syntax of programming languages.

First, we define the set of finite labelled trees.

Definition 2.17 (Finite Labelled Trees)

Assume the basic elements are trees l_1, \dots, l_m consisting of just one node (a leaf). To build more trees, we use constructors c_1, \dots, c_k , each with an associated arity. Given trees t_1, \dots, t_{n_i} , the constructor c_i of arity n_i builds the tree $c_i(t_1, \dots, t_{n_i})$, which has a root labelled c_i and subtrees t_1, \dots, t_{n_i} .

The set of finite labelled trees is the smallest set that contains all the basic trees l_1, \dots, l_m and is closed under the constructor operators.

The constructors could also have other parameters, for instance, we could also include numbers or strings to be stored at each node of the tree (see the exercises at the end of this chapter).

Before stating the induction principle for trees, let us give an example.

Example 2.18

If we have just one kind of leaf l , and only one constructor, of arity 2, let us call it *btree*, we can build the following binary trees:

- l : a leaf is a tree;
- $\text{btree}(l, l)$: this tree has just a root node and two leaves;
- $\text{btree}(\text{btree}(l, l), l)$ and $\text{btree}(l, \text{btree}(l, l))$: these are the only trees with three leaves;
- $\text{btree}(\text{btree}(l, l), \text{btree}(l, l))$: this tree has a root, two internal nodes, and four leaves.

The induction principle follows the inductive definition of tree (Definition 2.17).

Definition 2.19 (Principle of Induction for Trees)

To prove a property P for all finite labelled trees, it is sufficient to show:

- *Basis*: $P(l_1), \dots, P(l_m)$.

This means that we need to prove that the property holds for every basic tree.

- *Induction Step*:

For each tree constructor c_i (with $n_i \geq 1$ arguments), if the property holds for any trees t_1, \dots, t_{n_i} then it holds for $c_i(t_1, \dots, t_{n_i})$. More precisely:

$$\forall c, t_1, \dots, t_n. (P(t_1) \wedge \dots \wedge P(t_n) \Rightarrow P(c(t_1, \dots, t_n)))$$

It means that, for each tree constructor c of arity n , we need to prove the property for the tree $c(t_1, \dots, t_n)$ under the assumption that the property holds for the trees t_1, \dots, t_n .

In the induction step, $P(t_1), \dots, P(t_n)$ are the induction hypotheses.

Similarly, to define a function on all the elements of a set defined by structural induction, it is sufficient to define it for the basic elements, and to give a recursive definition for the elements defined by constructors. More precisely, to define function f on a set defined by structural induction from the basic elements l_1, \dots, l_m , using constructors c_1, \dots, c_k , where c_i has arity n_i , we need to define:

- $f(l_i)$ for each i such that $1 \leq i \leq m$, and
- $f(c_i(t_1, \dots, t_{n_i}))$, for each constructor c_i ($1 \leq i \leq k$) and arbitrary elements t_1, \dots, t_{n_i} ; in the definition of $f(c_i(t_1, \dots, t_{n_i}))$ we may use $f(t_j)$ for any $1 \leq j \leq n_i$.

For example, the function that computes the number of leaves in a binary tree (see Example 2.18), can be defined as follows.

- $leaves(l) = 1$
- $leaves(btree(t_1, t_2)) = leaves(t_1) + leaves(t_2)$.

Since programs are represented as abstract syntax trees, the principle of induction we have just given for finite labelled trees can be applied when we need to prove that all the programs written in a given programming language satisfy a certain property. For this, it is sufficient to specialise the induction principle to take into account the specific sets of basic elements and constructors in the abstract syntax under consideration.

Example 2.20

Let us consider the particular case of arithmetic expressions (which are part of all general purpose programming languages). We have already given the abstract syntax of arithmetic expressions in Example 1.2: the grammar provided defines the set of trees representing arithmetic expressions. There are two cases, corresponding, respectively, to tokens representing numbers and expressions defined by applying a binary operator to two subtrees. Thus, the set of labelled trees corresponding to

arithmetic expressions can be built inductively: the basic trees are built using number tokens, and the inductive trees are built using four constructors, one for each of the four binary operators $+$, $*$, $-$, div in our language.

To prove that a property P holds for all the abstract syntax trees corresponding to arithmetic expressions in the language, we proceed as follows:

1. Basis: Prove $P(n)$ for all number tokens n .
2. Induction Step: For each operator $Op \in \{+, *, -, div\}$, and each pair E, E' of abstract syntax trees of arithmetic expressions, prove that $P(E)$ and $P(E')$ implies $P(Op(E, E'))$.

Rule Induction. All the induction principles we discussed so far are in fact particular cases of a more general mechanism. We will now give a definition of an inductive set as a subset of a universal set characterised by its generators, that is, a subset defined by

- a finite set of basic elements and
- a finite set of rules that specify how to generate elements of the subset.

Any set of objects can be used as a universal set. Once we have fixed our universal set T , we can define the basic elements of the inductive subset by enumerating them, and define the generators by showing how new elements are constructed in terms of previously defined objects. Formally, we will write inductive definitions using *axioms* to represent the basic elements (the *basis* of the inductive definition) and *rules* to define the way constructors work.

We will then give an induction principle associated with the definition of the inductive subset. The axioms will provide the basis for the induction, and the rules will represent the induction step. We will give examples below, but first let us define more precisely the notions of axiom and rule.

Definition 2.21 (Axioms and Rules)

Let T be a set of objects (i.e. our universal set).

An axiom is simply an element of T . The standard notation for axioms is:

$$\frac{}{t}$$

A rule is a pair (H, c) where

- H is a non-empty subset $\{h_1, \dots, h_n\}$ of T , called the *hypotheses* or *premises* of the rule;
- c is an element of T , called the *conclusion* of the rule.

The standard notation for rules is:

$$\frac{h_1 \quad \dots \quad h_n}{c}$$

Rules may have conditions, which we write on the side (we give examples below).

We now define precisely the inductive set specified by a collection of axioms and rules.

Definition 2.22 (Inductive Set Specified by Axioms and Rules)

The subset I of T inductively defined by a collection A of axioms and R of rules is the smallest subset of T that includes the set A of axioms and is closed under all the rules in R .

According to this definition, the inductive set I consists of those elements t in T such that

- $t \in A$ (that is, t is an axiom), or
- there are $t_1, \dots, t_n \in I$ and a rule $(H, c) \in R$ such that $H = \{t_1, \dots, t_n\}$ and $t = c$.

We can apply this technique to define the inductive sets we have already studied.

Example 2.23

We can define the set of natural numbers using one axiom:

$$\frac{}{0}$$

and one rule:

$$\frac{n}{n+1}$$

The set of lists of natural numbers can be defined by the axiom:

$$\frac{}{nil}$$

and the rule:

$$\frac{l}{cons(h, l)} \text{ if } h \in \mathcal{N}$$

Note the condition on the rule, which specifies that h is a natural number.

Another example is the definition of the set of strings that have an even number of characters from an alphabet Σ . In this case the universal set is the set of all the sequences of characters in Σ . Let us denote the empty string by ϵ . We can define the subset of strings whose length is an even number using just the axiom

$$\frac{}{\epsilon}$$

and a set of rules of the form

$$\frac{s}{a s b} \text{ if } a, b \in \Sigma$$

According to this rule, we can build a string with an even number of characters using a previously built string s , which has an even number of characters, and two arbitrary characters a and b in Σ ; asb denotes the string obtained by adding a at the beginning of s and b at the end. We will often omit ϵ when there is no confusion, so we will write ab instead of $a\epsilon b$.

We can think of this as a rule scheme: there is one rule for each pair of characters.

To show that an element t of T is in the inductive set I it is sufficient to show that t is an axiom, or that there is a way of generating t starting from the axioms and using the rules. The latter can be shown by building a *proof tree* for t .

Definition 2.24 (Proof tree)

Given an inductive definition of I using axioms A and rules R , a proof tree for t has root, also called *conclusion*, t . Its leaves correspond to axioms in A , and for each non-leaf node t_i there is a rule $(\{t_{i1}, \dots, t_{im_i}\}, t_i)$ in R relating t_i and its children t_{i1}, \dots, t_{im_i} in the proof tree.

This kind of proof tree is usually written:

$$\begin{array}{c}
 \begin{array}{ccc} \vdots & & \vdots \\ \hline t_{11} & \dots & t_{1m_1} \end{array} & & \begin{array}{ccc} \vdots & & \vdots \\ \hline t_{n1} & \dots & t_{nm_n} \end{array} \\
 \hline
 t_1 & \dots & t_n \\
 \hline
 t
 \end{array}$$

For example, the proof tree for the number 2, using the inductive definition of the set of natural numbers given in Example 2.23, is:

$$\begin{array}{c}
 \hline
 0 \\
 \hline
 1 \\
 \hline
 2
 \end{array}$$

Similarly, we can build a proof tree for the string $abab$:

$$\begin{array}{c}
 \hline
 \epsilon \\
 \hline
 ba \\
 \hline
 abab
 \end{array}$$

In the following example, we use axioms and rules to give an inductive definition of an evaluation relation for arithmetic expressions. Here, the universal set is the set of pairs where the first element is an abstract syntax tree representing an arithmetic expression, and the second a value (an integer number in this case).

Example 2.25 (Inductive Definition of Evaluation)

Consider the abstract syntax of arithmetic expressions given in Example 1.2. We define now, using axioms and rules, a binary relation that associates the (abstract syntax tree of an) arithmetic expression E with a number n . We will write $E \Downarrow n$ to indicate that the expression E evaluates to the number n .

Axioms.

A number is already a value. Therefore for each number we need an axiom: $0 \Downarrow 0$, $1 \Downarrow 1$, $2 \Downarrow 2$, etc. Instead of writing these explicitly, we will use an *axiom scheme*:

$$n \Downarrow n$$

which represents the infinite set of axioms obtained by replacing n with a number.

From now on, we will not distinguish between axioms and axiom schemes. We will simply write $n \Downarrow n$, with the assumption that n is any number.

Rules.

Similarly, we will need rules for each operator $Op \in \{+, *, -, div\}$, which we represent as *rule schemes*:

$$\frac{E_1 \Downarrow n_1 \quad E_2 \Downarrow n_2}{Op(E_1, E_2) \Downarrow n} \text{ if } n = (n_1 \text{ Op } n_2)$$

The condition “if $n = (n_1 \text{ Op } n_2)$ ” on the rule simply states that the number n in the conclusion of the rule is the result of the operation denoted by Op on the values n_1 and n_2 .

From now on, rule schemes will simply be called rules.

Associated to inductive definitions using axioms and rules, there is a principle of induction, called the *Principle of Rule Induction*.

Let I be a set defined by induction with axioms and rules (A, R) . To show that a property P holds for each element of the set I , that is, to prove

$$\forall i \in I. P(i)$$

it is sufficient to prove:

- *Basis*: $\forall a \in A. P(a)$
- *Induction Step*:
 $\forall (\{h_1, \dots, h_n\}, c) \in R. P(h_1) \wedge \dots \wedge P(h_n) \Rightarrow P(c).$

Thus, every proof by rule induction has two parts. In the first part, the property is proved for the axioms. In the second part, for each rule, assuming that the property is true for the premises of the rule (this is the induction hypothesis), the property is proved for the conclusion of the rule.

Example 2.26

We can prove using rule induction that each arithmetic expression has a unique value under the evaluation relation given in Example 2.25. In other words, the relation \Downarrow is

a function, and it is defined for every expression (we assume the arithmetic operators are total).

Basis: The basis of the induction holds trivially, since numbers have unique values (according to the axioms, the value of an expression n is the number n itself).

Induction Step: For non-atomic expressions, we remark that the value of an arithmetic expression of the form $Op(E_1, E_2)$ is uniquely determined by the values of its arguments E_1 and E_2 , which are unique by the induction hypotheses.

2.5 Further Reading

Makinson's book [1] provides a gentle introduction to logic, set theory and induction. Harper's book [2] and Winskel's book [3] include chapters on induction with a variety of examples, and Dowek's book [4] provides a formal definition of an inductive set as a least fixed point, including concise explanations of the notions underlying the definition.

2.6 Exercises

1. Show the following properties of the intersection and union operators on sets:

$$\begin{aligned}
 A \cup B &= B \cup A \\
 (A \cup B) \cup C &= A \cup (B \cup C) \\
 A \cup A &= A \\
 A \cap B &= B \cap A \\
 (A \cap B) \cap C &= A \cap (B \cap C) \\
 A \cap A &= A
 \end{aligned}$$

2. Show that for every formula ϕ in first-order logic, and every choice of universe U , valuation σ and interpretations for function and predicate symbols, the interpretations of ϕ and $\neg(\neg\phi)$ coincide, that is, double negation does not change the truth value of a formula.
3. Prove, by mathematical induction, that for any natural number n :

$$\begin{aligned}
 - \sum_{i=1}^n i &= n(n+1)/2 \\
 - \sum_{i=1}^n 2^{i-1} &= 2^n - 1
 \end{aligned}$$

4. Describe the set of trees generated by the following inductive definition:
Nil is a tree;
 If n is a number and t_1 and t_2 are trees, then $Tree(n, t_1, t_2)$ is a tree.
5. Prove that the principle of structural induction corresponds to a particular application of the principle of mathematical induction.
6. Define the abstract syntax of first-order logic formulas using a grammar.
 Using structural induction, define for each formula its set of free variables and bound variables.

7. Define by rule induction the set E of even natural numbers, using $n+2$ to generate the even number after n . Prove by rule induction that $2n$ is in the set E , for every natural number n .
8. Let $R \subseteq A \times A$ be a binary relation on A . Define by rule induction the relation R^* , that is, the reflexive-transitive closure of R .
9. (†) Let I be a set defined by rule induction using axioms A and rules R . An alternative way of defining I is by taking the intersection of all the subsets of T that include A and are closed under R :

$$I = \bigcap \{J \subseteq T \mid A \subseteq J \text{ and } J \text{ } R\text{-closed}\}$$

Prove that both definitions are equivalent.

References

1. D. Makinson, *Sets, Logic and Maths for Computing*, 2nd edn., Undergraduate topics in computer science (Springer, Berlin, 2012)
2. R. Harper, *Practical Foundations for Programming Languages* (Cambridge University Press, Cambridge, 2013)
3. G. Winskel, *The Formal Semantics of Programming Languages*, Foundations of computing (MIT Press, Cambridge, 1993)
4. G. Dowek, *Proofs and Algorithms—An Introduction to Logic and Computability*, Undergraduate topics in computer science (Springer, Berlin, 2011)

Programming Languages and Operational Semantics

A Concise Overview

Fernández, M.

2014, IX, 209 p. 10 illus., Softcover

ISBN: 978-1-4471-6367-1