1 An Overview of Fundamental Proof Methods

JUST AS procedures are essential tools in programming languages for expressing computations, proof methods play a similarly essential role in expressing deductions. For a simple example of a proof method, consider one that is traditionally known as *modus ponens* (Latin for "mode that affirms"):

Given a conditional $(p \Rightarrow q)$ and its antecedent p, conclude q.

In this textbook, although we will discuss this and other basic methods (in Chapters 4 and 5), we treat them as primitives upon which *higher-level* proof methods are founded, much as higher-level programming languages are ultimately based on machine language instructions. The principal higher-level methods we will study are *equality and implication chaining, induction, case analysis, proof by contradiction*, and *abstraction/specialization*. In the following sections we preview some of the key aspects of these methods and their formulations in Athena before going on to outline the overall structure of the book.

1.1 Equality chaining

Of all the proof methods covered in this textbook, equality chaining is probably the one that is most widely used in computer science—and, for that matter, in many branches of mathematics, natural science, and engineering. In fact, readers will likely have used equality chaining many times already, albeit informally and perhaps without a thorough understanding of either its technical basis or the full extent of its applicability. As a brief but not completely trivial example of equality chaining, consider proving the algebraic identity

$$(a^{-1})^{-1} = a, \tag{1.1}$$

where we are given the following equations as axioms:

$$\begin{aligned} &\textit{Right-Identity}: && x \cdot I &&= x \\ &\textit{Left-Identity}: && I \cdot x &&= x \\ &\textit{Right-Inverse}: && x \cdot x^{-1} &&= I \end{aligned}$$

Here I is the *identity element* (or *neutral element*) for the operation $\cdot$. In the case where the domain over which x ranges is the set of nonzero real numbers and $\cdot$ is real-number multiplication, I would simply be 1. But these identities also hold in many other domains, for example, $n \times n$ invertible matrices over the reals, where $\cdot$ is matrix multiplication and I is the $n \times n$ matrix with 1's on its diagonal and 0's elsewhere.

Using these identities, we might write the following equality chain as a proof of (1.1):

$$\begin{aligned}(a^{-1})^{-1} &= I \cdot (a^{-1})^{-1} \\ &= (a \cdot a^{-1}) \cdot (a^{-1})^{-1} \\ &= a \cdot (a^{-1} \cdot (a^{-1})^{-1}) \\ &= a \cdot I \\ &= a\end{aligned}$$

Given this equality chain as proof, the reader may already be convinced of the validity of (1.1), but that impression might be something of an illusion based on familiarity (or overfamiliarity) with the somewhat informal way in which such proofs are typically presented in textbooks or journal articles. In almost all of the proofs presented or assigned as exercises in this textbook, we will adopt a more explicit style of proof that yields a higher level of clarity, but without departing drastically from the usual textbook or journal proof style.

To preview some of the key aspects of this style, let us take a closer look at what is involved in the above proof and show where we would revise it to be clearer and more complete. As a first step, let us indicate which of the given identities is used in each step of the chain by annotating the step with the identity's label.

$$\begin{aligned}(a^{-1})^{-1} &= I \cdot (a^{-1})^{-1} && [\textit{Left-Identity}] \\ &= (a \cdot a^{-1}) \cdot (a^{-1})^{-1} && [\textit{Right-Inverse}] \\ &= a \cdot (a^{-1} \cdot (a^{-1})^{-1}) && [?] \\ &= a \cdot I && [\textit{Right-Inverse}] \\ &= a && [\textit{Right-Identity}]\end{aligned}$$

We see that the third step does not correspond to any of the given identities. We might gloss over this point by saying that that step is just a "regrouping" of the multiplication operations, but in fact the validity of such a regrouping depends on yet another identity, namely

$$\textit{Associativity}\text{: } (x \cdot y) \cdot z = x \cdot (y \cdot z),$$

which does hold, for example, for multiplication of reals and for multiplication of square matrices. But associativity does not hold for all binary operators (e.g., consider real or integer subtraction). Assuming we are also given this identity, our annotated proof now becomes:

$$\begin{aligned}(a^{-1})^{-1} &= I \cdot (a^{-1})^{-1} && [\textit{Left-Identity}] \\ &= (a \cdot a^{-1}) \cdot (a^{-1})^{-1} && [\textit{Right-Inverse}] \\ &= a \cdot (a^{-1} \cdot (a^{-1})^{-1}) && [\textit{Associativity}] \\ &= a \cdot I && [\textit{Right-Inverse}] \\ &= a && [\textit{Right-Identity}]\end{aligned}$$

Next, if we look more closely at the first step of the chain, we see that what is used to justify this step is a particular *instance* of *Left-Identity*, namely

$$I \cdot (a^{-1})^{-1} = (a^{-1})^{-1},$$

in which the variable x of *Left-Identity* is specialized to the particular term $(a^{-1})^{-1}$. The differing roles of the variable x in *Left-Identity* and of a in the term $(a^{-1})^{-1}$ in the equation being derived can be confusing and sometimes lead to errors in reasoning. (This would be especially the case if we had written (1.1) as $(x^{-1})^{-1} = x$.) To clarify the role of the variables in the given identities, let us restate them showing explicitly that x, y, and z are *universally quantified* (over D):

$$\begin{aligned}
&\textit{Right-Identity}\colon \forall x : D \,.\, x \cdot I = x \\
&\textit{Left-Identity}\colon \quad \forall x : D \,.\, I \cdot x = x \\
&\textit{Right-Inverse}\colon \forall x : D \,.\, x \cdot x^{-1} = I \\
&\textit{Associativity}\colon \quad \forall x : D \;\; y : D \;\; z : D \,.\, (x \cdot y) \cdot z = x \cdot (y \cdot z)
\end{aligned}$$

Read $\forall\, x : D$ as "for all x in D" or "for every x in D."

So the instance of *Left-Identity* used in the first step of the equality chain is obtained by specializing its universally quantified variable x to the term $(a^{-1})^{-1}$. But what of the variable a in that term? We can clarify its role by beginning the proof with the phrase "Let a be an arbitrarily chosen element (of D)." So the full proof reads:

To prove $\forall\, a : D \,.\, (a^{-1})^{-1} = a$, let a be an arbitrarily chosen element of D. Then:

$$\begin{aligned}
(a^{-1})^{-1} &= I \cdot (a^{-1})^{-1} && [\textit{Left-Identity}] \\
&= (a \cdot a^{-1}) \cdot (a^{-1})^{-1} && [\textit{Right-Inverse}] \\
&= a \cdot (a^{-1} \cdot (a^{-1})^{-1}) && [\textit{Associativity}] \\
&= a \cdot I && [\textit{Right-Inverse}] \\
&= a && [\textit{Right-Identity}]
\end{aligned}$$

In the second step of the chain, there is another aspect of the use of the identity *Right-Inverse*, namely that the instance $a \cdot a^{-1} = I$ of $\forall\, x : D \,.\, x \cdot x^{-1} = I$ is used to replace only a *subterm* of $(a \cdot a^{-1}) \cdot (a^{-1})^{-1}$, not the whole term. What justification is there for such a subterm replacement? The answer to that question is one of the points about equality chaining that we discuss in detail in Chapter 3. For now, we might consider further annotation of equality chain steps to identify the subterm being replaced, and perhaps also the variable substitutions being made in the justifying identity. While there are established notations for doing so, requiring their use would make equality chains very tedious to write, and the extra detail could hinder readability. But once the underlying principles are understood, equality chains can remain quite readable and convincing without the extra detail.

For mechanical checking of equality chain proofs, both the identification of the subterm being replaced and the variable substitution involved can be done quite efficiently with

limited subterm-search and term-matching capabilities. The Athena system used in this textbook provides these capabilities, among many other places, in its `chain` method, so that our example proof can be expressed at virtually the same level of explicitness as before (writing $*$ for $\cdot$ and `(inv a)` for a^{-1}):

```
1 conclude (forall ?a:D . inv (inv ?a) = ?a)
2   pick-any a:D
3     (!chain [(inv (inv a))
4            = (I * inv (inv a))             [Left-Identity]
5            = ((a * inv a) * inv (inv a))   [Right-Inverse]
6            = (a * ((inv a) * inv (inv a))) [Associativity]
7            = (a * I)                       [Right-Inverse]
8            = a                             [Right-Identity]])
```

Much more on using `chain` to prove equations in Athena can be found in Chapter 3, where there are many examples and related exercises. We can also use `chain` to prove implications and equivalences. We give a couple of examples of implication chaining in Section 1.3, discuss it and equivalence chaining in detail in Chapter 6, and use them both extensively in later chapters.

Note that function applications in Athena are written in prefix notation as $(f\ a_1 \cdots a_n)$, that is, the function symbol f is written first, followed by the arguments $a_1 \cdots a_n$, which are separated simply by white space, with the whole thing surrounded by an outer pair of parentheses. However, binary function symbols can be used in infix, and different precedence levels can be given to different symbols, or even to the same symbol in different contexts, and this can greatly reduce notational clutter. In addition, parentheses can be omitted when writing successive applications of a unary function symbol such as `inv`. For example, we could have written `(inv inv a)` instead of `(inv (inv a))` in the preceding proof. In fact, the notation of the proof can be simplified further if we ensure that `inv` binds tighter than the operation `*`, in which case the term in line 6 could be written simply as `(a * inv a * inv inv a)`, since every function symbol is right-associative by default. However, in the final version that follows we don't go that far, since terms are sometimes more readable if a few parentheses are used even when not strictly necessary.

The following is a self-contained Athena module containing all the code needed for this example, including the introduction of the domain D and the declarations of the relevant function symbols and their precedence levels.[1] The pound sign # begins a comment.

```
module M {
  domain D
  declare I: D
  declare *: [D D] -> D [200] # We give a precedence of 200 to *
  declare inv: [D] -> D [220] # and 220 to inv
```

1 A module M, introduced with the syntax **module** M { ... }, is essentially a namespace containing sorts, function symbols, definitions, and possibly other nested submodules. We put this code into a module mostly because it allows us to declare the symbol $*$ distinctly from its predefined meaning in Athena (as numeric multiplication).

```
  # Let's define some handy abbreviations for a few variables over D:
  define [x y z] := [?x:D ?y:D ?z:D]

  # We can now assert the four given axioms:
  assert* Left-Identity  := (I * x = x)
  assert* Right-Identity := (x * I = x)
  assert* Right-Inverse  := (x * inv x = I)
  assert* Associativity  := (x * (y * z) = (x * y) * z)

  # Finally, the desired proof:
  conclude (forall x . inv inv x = x)
    pick-any a:D
      (!chain [(inv inv a) = (I * inv inv a)             [Left-Identity]
                           = ((a * inv a) * inv inv a)   [Right-Inverse]
                           = (a * (inv a * (inv inv a))) [Associativity]
                           = (a * I)                     [Right-Inverse]
                           = a                           [Right-Identity]])
} # close module M
```

The directive **assert** is used to insert a sentence into the current *assumption base*, while at the same time possibly giving the sentence a name for future reference (the identifier preceding the symbol :=). We will have much more to say about assumption bases later on, but for now you can think of the assumption base as the set of our working assumptions (axioms), as well as any results (theorems) we have managed to derive from those assumptions. The directive **assert*** works just like **assert**, except that it first *closes* the given sentence p; or, more precisely, it inserts into the assumption base a copy of p that is universally quantified over all of p's free variables. Accordingly, **assert*** `(I * x = x)` is equivalent to **assert** `(forall x . I * x = x)`. This shorthand saves us from having to universally quantify our axioms manually.

Before we leave the topic of equality chaining, let us consider another important aspect of such proofs: starting with the equation $s = t$ to be proved, and letting $s_0 = s$, how do we find the right sequence of terms $s_1, s_2, \ldots, s_n = t$ to link together? Of course, we are restricted in this search in the first place by the set of available identities. Even so, it is not always easy to see how to get started. In the preceding proof, for example, the first and second steps replace the current term by a larger term (i.e., $(a^{-1})^{-1}$ by $I \cdot (a^{-1})^{-1}$ in the first step, I by $a \cdot a^{-1}$ in the second step), even though the goal term a is smaller than the starting term $(a^{-1})^{-1}$. Even if the given equality chain convincingly proves the desired equation, the chain by itself does not explain how its author went about finding the right terms to link together—it may seem to have required a level of creativity that goes well beyond just the mechanics of writing proofs that are logically correct.

One approach to constructing equality chains that requires less creativity is to try to find a chain of a special structure: working from both sides, s and t, try to *reduce them to a common term*. We discuss this strategy in detail in Chapter 3 and recommend attempting

it in many cases. However, this reduction strategy is not always applicable; for example, it does not work for proving $(a^{-1})^{-1} = a$ from the given identities. In Section 1.5, we preview another strategy that can be used to limit the number of occasions in which difficult searches for a proof are required.

Before leaving this section, consider the following problem involving three people: Jack, Anne, and George.

> Jack is looking at Anne, and Anne is looking at George. Jack is married, George is not. Is some married person looking at an unmarried person?

Take a moment to try to answer the question, either (A) yes, (B) no, or (C) cannot be determined from the given information. We give the answer in a later section of this chapter. (This problem has nothing to do with proving equalities; we insert it here only to give the reader a chance to try answering before seeing the solution discussed later.)

1.2 Induction

Not every valid equation can be proved just by chaining together instances of other valid equations. Proving an equation may require some form of *mathematical induction*. We say "some form of" because there are many manifestations of this important proof method, including *ordinary* and *strong* induction, and several variations thereof for different data types. Mathematical induction can also be used to prove other kinds of sentences besides equations, but we begin its study in Chapter 3 with applications to equations expressing properties such as associativity of the usual addition and multiplication operators on natural numbers. The value of these proof examples and exercises is not so much that they are needed in practice—other, higher-level methods of assuring correctness of basic arithmetic are usually preferable—but as training for formulating and proving similar properties of other abstract data types, like lists or trees. In Chapter 3 we introduce a mathematical induction principle for linear lists, and in Chapter 8, we do the same for binary trees.

For natural numbers, ordinary mathematical induction takes the form of dividing a proof of $\forall\, n\,.\, P(n)$ into two cases: (i) $P(0)$ and (ii) $\forall\, n\,.\, P(n) \Rightarrow P(n+1)$. Case (i) is called the *basis case*, case (ii) is called the *induction step*, and within it, the antecedent $P(n)$ is called the *induction hypothesis*. Proof of the basis case and the induction step suffices, basically because every natural number is either 0 or can be constructed from 0 by a finite number of increments by 1. We make these statements precise in Chapter 3, but for now we can preview how we deal with induction proofs in Athena by defining natural numbers as an *algebraic datatype* `N`:

```
datatype N := zero | (S N)
```

The symbols `zero` and `S` are the so-called *constructors* of `N`. If we interpret `S` as incrementing by 1 (and thus justly call it the "successor" function), we can read this recursive

datatype[2] declaration as saying in Athena what we just said in English, that every natural number is either 0 or can by constructed from 0 by a finite number of increments by 1. Now let's look at a simple example of proving a property of natural-number addition, where the proof requires induction. We declare and axiomatize the addition function, `Plus`, as follows:

```
declare Plus: [N N] -> N [+]

define [n m] := [?n:N ?m:N]

assert* Plus-zero-axiom := (n + zero = n)
assert* Plus-S-axiom := (n + S m = S (n + m))
```

In the declaration, the annotation `[+]` means that we *overload* the built-in symbol `+` to designate `Plus` whenever the arguments of `+` are terms of sort `N`. The axioms express the identities $n+0=n$ and $n+(m+1)=(n+m)+1$. Suppose now that we want to prove the identity $0+n=n$, which in Athena we express as follows:

```
define Plus-S-property := (forall n . zero + n = n)
```

Of course, if we had already proved that `Plus` is commutative (the identity $n+m=m+n$), then we could prove `Plus-S-property` with a simple equation chain:

```
pick-any n:N
  (!chain [(zero + n) = (n + zero) [Plus-commutative]
                      = n          [Plus-zero-axiom]])
```

However, we will see in Chapter 3 that to prove commutativity of `Plus` (by induction) we need to use `Plus-S-property` in the proof! So, to avoid circular reasoning, we need to prove `Plus-S-property` without using commutativity. We must use induction, but within each case of the induction the proof is just a simple equation chain using the axioms and, in the induction step, the induction hypothesis. Here is the proof:

```
by-induction Plus-S-property {
  zero => (!chain [(zero + zero) = zero [Plus-zero-axiom]])
| (n as (S m)) =>
    conclude (zero + n = n)
    # The induction-hypothesis is already in the assumption base.
    # Here we just give it a name:
      let {induction-hypothesis := (zero + m = m)}
        (!chain [(zero + S m)
                 = (S (zero + m))      [Plus-S-axiom]
                 = (S m)               [induction-hypothesis]])
}
```

2 We use the phrase "data type" when we talk about data types in general, but the single word "datatype" when we are specifically discussing an algebraic datatype, particularly in the context of an Athena **`datatype`** definition.

In computation, we rarely represent numbers in the *unary* form of applications of `S` to `zero`—sometimes also called a *Peano* representation after the Italian mathematician Giuseppe Peano (1858–1932), one of the founders of mathematical logic and set theory—preferring instead a binary, octal, decimal, or hexadecimal representation. But for rigorously specifying and proving correctness of certain numeric or seminumeric algorithms, the Peano representation and mathematical induction principles based on it have the advantage of simplicity. For example, in Chapter 3 we define an exponentiation operator with equations $x^0 = 1$ and $x^{n+1} = x \cdot x^n$ and use mathematical induction to prove a few of its properties. But, more importantly, in Chapter 12 we take this simple definition as the *specification* of the meaning of x^n, against which a more efficient algorithm for computing x^n is proved correct. That algorithm reduces the problem for $n > 0$ to one for $n/2$ (instead of reducing it from $n + 1$ to n), so the proof is done with the form of mathematical induction called *strong induction*: to prove $\forall\, n\, .\, P(n)$, prove

$$\forall\, n\, .\, [\forall\, k\, .\, k < n \Rightarrow P(k)] \Rightarrow P(n).$$

Here, $[\forall\, k\, .\, k < n \Rightarrow P(k)]$ is called the *strong induction hypothesis*. There are several subtle but important points to understand about strong induction and its applications (e.g., why there is no apparent need for a basis case), as discussed in Chapter 12.

Similarly, as we progress to induction principles for abstract data types like linear lists and binary trees, we must understand new twists that distinguish them from the natural number versions. In proofs of binary tree properties, for example, we must formulate and take advantage of *two* induction hypotheses, one for each of the left and right subtrees of a nonempty tree. We will see examples of this in Section 8.4.

1.3 Case analysis

What was your answer to the little problem posed at the end of Section 1.1? If it was C ("cannot be determined from the given information"), you are in good company—but wrong. The problem is from an article in *Scientific American* [93] that discussed why people often fail to find the correct solution to various logic puzzles. In this case, C was chosen more than 80 percent of the time, apparently because solvers didn't take time to go through all of the possibilities carefully. The correct answer is A ("yes"):

> Either Anne is married or she isn't. If she is married, then since she is looking at George, and George is not married, we have a married person looking at an unmarried person. If Anne is unmarried, then since Jack, a married person, is looking at Anne, we know in this case too that a married person is looking at an unmarried one.

The form of reasoning used here, known as *case analysis*, is a powerful proof method, as we will see in many examples and exercises in this textbook. However, the kinds of applications we will discuss, beginning in Chapter 4, will *not* be simple logic puzzles like the one above, since our goal in this textbook is to develop experience with logic and proof methods in applications that are most relevant in computer science.

Thus, we will not be coming back to this or other examples of logic puzzles. Nonetheless, you might be curious about how you could set up the problem and derive the answer in Athena. The following is a complete formulation; of course, it depends on features of Athena that are only covered later, such as implication chaining, but if you compare its basic structure with the English version above, you will see a strong correspondence.

```
module M {

  domain Person

  declare married: [Person] -> Boolean
  declare looking-at: [Person Person] -> Boolean
  declare Jack, Anne, George: Person

  define [p1 p2] := [?p1:Person ?p2:Person]

  assert [(Jack looking-at Anne)
          (Anne looking-at George)
          (married Jack)
          (~ married George)]

# Is some married person looking at an unmarried person?
# Yes, as shown with a case analysis:

  conclude goal := (exists p1 p2 . married p1 &
                                   p1 looking-at p2 &
                                   ~ married p2)
    (!two-cases
      assume case1 := (married Anne)
        (!chain-> [case1
               ==> (married Anne &
                    Anne looking-at George
                    & ~ married George)      [augment]
               ==> goal                      [existence]])
      assume case2 := (~ married Anne)
        (!chain-> [case2
               ==> (married Jack &
                   Jack looking-at Anne &
                   ~ married Anne)           [augment]
               ==> goal                      [existence]]))
}
```

(Read the symbol ==> as "implies," & as "and," and ~ as "not." These and other sentential connectives used in Athena are discussed in Section 2.4, along with their correspondence to other commonly used symbols.)

We will see many examples of case analysis in proofs about algorithms and data structures, in many variations. More general methods of breaking a proof down into two or more cases, independently of any datatype, are introduced in Chapter 4.

1.4 Proof by contradiction

When proving a negative result, often the most natural method to apply is *proof by contradiction*: assume the opposite, and show that that assumption leads to a contradiction. As an example, consider a relation $<$ on some domain D, with the properties

$$\textit{Irreflexivity}\colon\ \forall\, x\!:\!D\ .\ \sim x < x$$
$$\textit{Transitivity}\colon\ \forall\, x\!:\!D\ \ y\!:\!D\ \ z\!:\!D\ .\ x < y \wedge y < z \Rightarrow x < z$$

Then the $<$ relation also has the property

$$\textit{Asymmetry}\colon\ \forall\, x\!:\!D\ \ y\!:\!D\ .\ x < y \Rightarrow \sim y < x$$

PROOF: By contradiction. Choose any a and b in D such that $a < b$ and assume, contrary to the stated conclusion, that $b < a$. Then from $a < b$, $b < a$, and transitivity, we obtain $a < a$. But, by irreflexivity, we have $\sim a < a$, and hence we have a contradiction. ■

The binary `by-contradiction` method is used to carry out such proofs in Athena. The first argument to this method is the sentence p we are trying to prove, and the second argument is a conditional of the form $(\overline{p} \Rightarrow \texttt{false})$, where $\overline{p}$ is the *complement* of p; that is, q if p is of the form `(~` q`)`, and `(~` p`)` otherwise. If this conditional is in the assumption base, then the result of this method application will be the goal p. And because deductions derive their conclusions and enter them into the assumption base, this second argument is typically expressed as a deduction. The most common way to derive `false` is via the `absurd` method, which takes two arguments, both of which must be in the assumption base, and the second of which is the negation of the first. For this example, we could set up the axioms and prove the theorem as follows:

```
module A {
  domain D
  declare <: [D D] -> Boolean

  define [x y z] := [?x:D ?y:D ?z:D]

  assert* irreflexivity := (~ x < x)
  assert* transitivity := (x < y & y < z ==> x < z)
```

```
  conclude asymmetry := (forall x y . x < y ==> ~ y < x)
    pick-any a:D b:D
      assume (a < b)
        (!by-contradiction (~ b < a)
          assume (b < a)
            let {less := (!chain-> [(b < a)
                                    ==> (a < b & b < a)    [augment]
                                    ==> (a < a)            [transitivity]]);
                 not-less := (!chain-> [true
                                        ==> (~ a < a)      [irreflexivity]])}
              (!absurd less not-less))
} # close module A
```

All of the details of this and similar proofs about a strict inequality relation are developed in Chapter 8 for the natural numbers, and in greater generality in Chapter 14.

Proof by contradiction can also be used to prove a positive result, provided that combining its negation with other known properties leads to a contradiction. Either way, the steps toward the contradiction may use any other proof method, including equality or implication chaining (as in the preceding example), case analysis, or even other proofs by contradiction. (Even a mathematical induction proof could be used along the way, but that would be best handled as a separate proof whose result is then used as a lemma in working toward the contradiction.)

As the asymmetry proof shows, a proof by contradiction can be done as a step in a larger proof. We will see many examples in which one or more proofs by contradiction are used within larger proofs, including cases where one is nested inside a larger proof by contradiction.

1.5 Abstraction/specialization

At the end of Section 1.1, we mentioned one strategy for proving equalities that required less creativity, namely "reduction to a common term," but we noted that it does not always work. There is another strategy that we advocate to limit the number of difficult proofs one has to write, not only when constructing equality chains but in many other cases as well: *Work at an abstract level*, proving theorems in the most general setting, then specializing them to concrete instances as needed. We already pointed out that the identities on which the proof of $(a^{-1})^{-1} = a$ depends are valid in a couple of important domains, $D \equiv$ nonzero real numbers and $D \equiv$ invertible $n \times n$ matrices. Going further, we may regard these identities as *axioms of an abstract structure* T, with which we are able to derive, using the constructed equality chain, the identity $(a^{-1})^{-1} = a$ as a theorem of T. Then for *any* concrete domain D in which the identities hold, the same proof can be reused to prove the corresponding concrete specialization of $(a^{-1})^{-1} = a$. The binary operator $\cdot$ might even

be specialized to an addition operator rather than a multiplication operator in a concrete domain. For example, let D be the integer domain Z, and specialize $\cdot$ to integer addition, I to its neutral element, and the inverse operator to integer negation, which we write as $+$, 0, and (unary) $-$, respectively. Then the given identities become

$$\begin{aligned}
&\textit{Right-Identity}: \forall\, x{:}Z\,.\, x + 0 = x\\
&\textit{Left-Identity}: \quad \forall\, x{:}Z\,.\, 0 + x = x\\
&\textit{Right-Inverse}: \forall\, x{:}Z\,.\, x + (-x) = 0\\
&\textit{Associativity}: \quad \forall\, x{:}Z\, y{:}Z\, z{:}Z\,.\, x + (y + z) = (x + y) + z
\end{aligned}$$

and the proof of $(- - a = a)$ becomes: Let a be an arbitrarily chosen element of Z. Then

$$\begin{aligned}
(- - a) &= (0 + - - a) && [\textit{Left-Identity}]\\
&= ((a + -a) + - - a) && [\textit{Right-Inverse}]\\
&= (a + (-a + - - a)) && [\textit{Associativity}]\\
&= (a + 0) && [\textit{Right-Inverse}]\\
&= a && [\textit{Right-Identity}]
\end{aligned}$$

But we don't have to construct this proof from scratch! Once we have found the abstract-level proof, we can store it along with the abstract theorem it proves, to be simply reused, with appropriate specialization, in any concrete domain (like Z) in which the axioms hold. We will see that Athena fully supports this strategy.

In mathematics, the advantages of working at an abstract level have long been known (they were conclusively demonstrated by mathematicians who were developing the field of *abstract algebra* over one hundred years ago). In computer science, the same benefits are seen in some textbooks and journal articles in which a theory is developed at an abstract level and then specialized in different ways. But readers with less mathematical training might be more familiar with how similar principles have also been applied in *computer programming*, not for constructing proofs, but for creating—automatically, during compilation—concrete algorithms or data structures as instances of abstract algorithms or data structures (often called *generic* algorithms or data structures).[3] In this book we will demonstrate the advantages of working at an abstract level by doing proofs in abstract theories and specializing them to various concrete domains, including examples very similar to the ones introduced in this section, starting in Chapter 14. Moreover, we will also apply this strategy to prove correctness properties of abstract algorithms in Chapters 15 and 16, including ones that are very similar to abstract algorithms in the C++ Standard Template Library.

3 Major languages that support abstract programming (or "generic programming"), to one degree or another, include Ada, C++, C#, and Java.

1.6 The usual case: Proof methods in combination

In this introductory overview, we have sampled several of the most important proof methods: equality chaining, induction, case analysis, proof by contradiction, and abstraction and specialization. In simple cases, a proof can be done with only one of these methods, but most proofs require combinations of them. A key goal of this textbook is to develop skills in choosing and properly applying a suitable combination of proof methods, particularly when proving a correctness or optimization property of an algorithm or data structure. We will see that, in many cases, a method that can be successfully applied to advance a proof is strongly suggested by the syntactic structure of the current proof goal and/or the current set of premises. Chapter 4 especially explores this line of attack. In other cases, it is the semantic content of the proof goal, or of available axioms and theorems, that provides important clues as to how to proceed. In such cases, it can be much more difficult to discern the right proof strategy than in the syntactically driven cases. We begin discussing how one can build up one's experience with such strategies in Section 3.11, and we return to this important topic repeatedly in the rest of the textbook. But indeed, in all cases, whether syntactically or semantically driven, there is no substitute for experience. Accordingly, throughout the book we have provided an abundance of proof examples and exercises.

1.7 Automated proof

Athena is integrated with a number of powerful automated theorem-proving systems (ATPs) and model-building systems such as SMT and SAT solvers.[4] Using these systems, all of the example proofs in this chapter could be performed automatically, in an entirely "push-button" manner. Having this level of automation available can be handy when tackling large verification projects, with Athena used to specify the system, test the specification, and outline the high-level structure of the proof, while ATPs fill in some of the more tedious proof details.

In this book, however, we make no use of ATPs. That decision was made for pedagogical reasons. We believe that the only way to gain sufficient experience in developing structured proofs is to write them from the ground up, without resorting to black-box oracles to cut corners or potentially get around tricky aspects of the required reasoning. Of course, building a proof "from the ground up" does not mean that the proof should be expressed exclusively in terms of low-level primitives such as modus ponens, in the same way that developing a computer program from the ground up does not mean that we should write the program in machine language. Abstraction mechanisms are indispensable in both cases. But, for pedagogical purposes, we believe that the internal workings of those abstraction

4 The integration is seamless in that these systems can be invoked as primitive methods similar to Athena's built-in inference methods, or as primitive procedures, and the interaction occurs at the level of Athena's polymorphic sort system; the details of the underlying translations are hidden.

mechanisms should be understandable in terms of the semantics of the host language, and that is not the case for external ATPs. That said, readers are encouraged to experiment with automated proof. To that end, we describe how ATPs can be used in Athena and provide a number of examples in Appendix D.

1.8 Structure of the book

This part continues with an introduction to the basics of Athena in Chapter 2. As noted in the Preface, not all of this chapter needs to be studied before going on; it can instead be treated as a reference while continuing through the rest of the book.

Part II (Fundamental Proof Methods) begins with an in-depth look at equality chaining and induction applied to natural numbers and lists, including natural-number addition, multiplication, and exponentiation, and list concatenation and reversal. Chapter 3 also introduces term evaluation and other tools that can be used before attempting proof, such as automated conjecture testing.

We then continue with sentential and first-order logic (Chapter 4 and Chapter 5, respectively), and generalize chaining to encompass implications and equivalences (Chapter 6).

In Part III (Proofs about Fundamental Datatypes), we begin by describing Athena's `module` mechanism for managing namespaces and for packaging large numbers of Athena proofs and programs into properly organized components (Chapter 7). In Chapter 8 we introduce and prove many simple properties of ordering relations over natural numbers. With these results at hand, we are able to formulate and prove properties of natural-number subtraction, which in later chapters are applied in defining integer arithmetic and natural-number division. The final sections of the chapter develop ordering properties of lists and binary search trees (over natural numbers).

In Chapter 9 we show how to develop the integer domain, including introduction of the important tool of homomorphic mappings between different representations to simplify proofs. The chapter concludes with a brief treatment of power series arithmetic.

Chapter 10 introduces some fundamental discrete structures, including ordered pairs, sets, relations, functions, and maps, and proves a number of useful theorems about them.

In Part IV (Proofs about Algorithms), the main emphasis is on correctness and optimization of algorithms, focusing again on operations on natural numbers. In Chapter 11, a binary-search algorithm provides the setting for discussion of whether initial correctness requirements placed on the behavior of the algorithm are sufficient to rule out unsatisfactory candidates. In Chapter 12, several variants of an exponentiation algorithm are formulated and proved correct. While each variant does some computation that is unnecessary, it is noted that this will be corrected in Chapter 15, in an abstract version that is also much more generally applicable. For the third example in this part, we extend the study of fundamental numeric properties to natural-number division, with proofs about