Mathematical Logic in Software Development Documentation

Release 1

Kevin Sullivan

Apr 09, 2018

CONTENTS:

1 1. Requirement, Specifications, and Implementations **1**

2 2. Logical Specifications, Imperative Implementations **3**
- 2.1 Imperative Languages for Implementations . . . 3
- 2.2 Declarative Languages for Specifications . . . 4
- 2.3 Refining Declarative Specifications into Imperative Implementations . . . 5
- 2.4 Why Not a Single Language for Programming and Specification? . . . 6

3 3. Problems with Imperative Code **7**

4 4. Pure Functional Programming as Runnable Mathematics **9**
- 4.1 The identify function (for integers) . . . 9
- 4.2 Data and function types . . . 10
- 4.3 Other function values of the same type . . . 10
- 4.4 Dafny is a Program Verifier . . . 12

5 5. Formal Verification of Imperative Programs **15**
- 5.1 Performance vs. Understandability . . . 15
- 5.2 Specification, Implementation, and Verification . . . 17
- 5.3 Declarative Input-Output Specifications . . . 17
- 5.4 Imperative Implementation . . . 19
- 5.5 Formal Verification . . . 19
- 5.6 Case Study: The Factorial Function . . . 19
- 5.7 Case Study: The Fibonacci Function . . . 23
- 5.8 What is Dafny, Again? . . . 25

6 6. Dafny Language: Types, Statements, Expressions **27**
- 6.1 Built-In Types . . . 27
- 6.2 Statements . . . 30
- 6.3 Expressions . . . 34

7 7. Set Theory **39**
- 7.1 Naive Set Theory . . . 39
- 7.2 Overly Naive Set Theory . . . 39
- 7.3 Sets . . . 40
- 7.4 Set Theory Notations . . . 41
- 7.5 Set Operations . . . 42
- 7.6 Tuples . . . 45
- 7.7 Relations . . . 46
- 7.8 Binary Relations . . . 46
- 7.9 Functions: *Single-Valued* Relations . . . 47

7.10 Properties of Functions . . . 48
7.11 Properties of Relations . . . 51
7.12 Sequences . . . 52

8 8. Boolean Algebra **53**
8.1 Boolean Algebra in Dafny . . . 53
8.2 Boolean Values . . . 54
8.3 Boolean Operators . . . 54
8.4 Formal Languages: Syntax and Semantics . . . 59
8.5 The Syntax of Boolean Expressions: Inductive Definitions . . . 59
8.6 The Semantics of Boolean Expressions: Recursive Evaluation . . . 62
8.7 The Syntax and Semantics of Programming Languages . . . 63

9 9. Propositional Logic **65**
9.1 Basic Terminology . . . 65
9.2 Propositional and Predicate Logic . . . 65
9.3 What is a Logic? . . . 66
9.4 Propositional Logic . . . 67
9.5 Inductive Definitions: The Syntax of Propositional Logic . . . 67
9.6 Semantics of Propositional Logic . . . 68
9.7 Inference Rules for Propositional Logic . . . 69
9.8 Using Logic in Practice . . . 70
9.9 Implementing Propositional Logic . . . 70
9.10 Satisfiability, Validity . . . 73
9.11 Logical Consequence . . . 73

10 10. Natural Deduction **75**

11 11. Sets as Domains: Predicate Logic **77**

12 12. Quantification: First-Order Logic **79**
12.1 Universal and Existential Quantification . . . 79
12.2 Introduction and Elimination Rules for Exists . . . 79
12.3 Introduction and Elimination Rules for Forall . . . 79
12.4 Induction Principles and Algebraic Data Types . . . 79

13 Indices and tables **81**

CHAPTER

ONE

1. REQUIREMENT, SPECIFICATIONS, AND IMPLEMENTATIONS

Software is an increasingly critical component of major societal systems, from rockets to power grids to healthcare, etc. Failures are not always bugs in implementation code. The most critical problems today are not in implementations but in requirements and specifications.

- **Requirements:** Statements of the effects that a system is meant to have in a given domain
- **Specification:** Statements of the behavior required of a machine to produce such effects
- **Implementation:** The definition (usually in code) of how a machine produces the specified behavior

Avoiding software-caused system failures requires not only a solid understanding of requirements, specifications, and implementations, but also great care in both the *validation* of requirements and of specifications, and *verification* of code against specifications.

- **Validation:** *Are we building the right system?* is the specification right; are the requirements right?
- **Verification:** *Are we building the system right?* Does the implementation behave as its specification requires?

You know that the language of implementation is code. What is the language of specification and of requirements?

One possible answer is *natural language*. Requirements and specifications can be written in natural languages such as English or Mandarin. The problem is that natural language is subject to ambiguity, incompleteness, and inconsistency. This makes it a risky medium for communicating the precise behaviors required of complex software artifacts.

The alternative to natural language that we will explore in this class is the use of mathematical logic, in particular what we call propositional logic, predicate logic, set theory, and the related field of type theory.

Propositional logic is a language of simple propositions. Propositions are assertions that might or might not be judged to be true. For example, *Tennys (the person) plays tennis* is actually a true proposition (if we interpret *Tennys* to be the person who just played in the French Open). So is *Tennys is from Tennessee*. And because these two propositions are true, so is the *compound* proposition (a proposition built up from smaller propositions) that Tennys is from Tennessee **and** Tennys plans tennis.

Sometimes we want to talk about whether different entities satisfy give propositions. For this, we introduce propositions with parameters, which we will call *properties*. If we take *Tennys* out of *Tennys plays tennis* and replace his name by a variable, *P*, that can take on the identify of any person, then we end up with a parameterized proposition, *P plays tennis*. Substituting the name of any particular person for *P* then gives us a proposition *about that person* that we can judge to be true or false. A parameterized proposition thus gives rise to a whole family of propositions, one for each possible value of *P*.

Sometimes we write parameterized propositions so that they look like functions, like this: *PlaysTennis(P)*. *PlaysTennis(Tennys)* is thus the proposition, *Tennys plays Tennis* while *PlaysTennis(Kevin)* is the proposition *Kevin plays Tennis*. For each possible person name, *P*, there is a corresponding proposition, *PlaysTennis(P)*.

Some such propositions might be true. For instance, *PlaysTennis(Tennys)* is true in our example. Others might be false. A parameterized proposition thus encodes a *property* that some things (here people) have and that others don't have (here, the property of *being a tennis player*).

A property, also sometimes called a *predicate*, thus also serves to identify a *subset* of elements in a given *domain of discourse*. Here the domain of discourse is the of all people. The subset of people who actually do *play tennis* is exactly the set of people, P, for whom *PlaysTennis(P)* is true.

We note briefly, here, that, like functions, propositions can have multiple parameters. For example, we can generalize from *Tennys plays Tennis **and** Tennys is from Tennessee** to *P plays tennis and P is from L,* where P ranges over people and L ranges over locations. We call a proposition with two or more parameters a *relation*. A relation picks out *combinations* of elements for which corresponding properties are true. So, for example, the *pair* (Tennys, Tennessee) is in the relation (set of *P-L* pairs) picked out by this parameterized proposition. On the other hand, the pair, (Kevin, Tennessee), is not, because Kevin is actually from New Hampshire, so the proposition *Kevin plays tennis **and** Kevin is from Tennessee** is not true. More on relations later!

CHAPTER

TWO

2. LOGICAL SPECIFICATIONS, IMPERATIVE IMPLEMENTATIONS

We've discussed requirements, specifications, and implementations as distinct artifacts that serve distinct purposes. For good reasons, these artifacts are usually written in different languages. Software implementations are usually written in programming languages, and, in particular, are usually written in *imperative* programming languages. Requirements and specifications, on the other hand, are written either in natural language, e.g., English, or in the language of mathematical logic.

This unit discusses these different kinds of languages, why they are used for different purposes, the advantages and disadvantages of each, and why modern software development requires fluency in and tools for handling artifacts written in multiple such languages. In particular, the educated computer scientist and the capable software developer must be fluent in the language of mathematical logic.

2.1 Imperative Languages for Implementations

The language of implementations is code, usually written in what we call an *imperative* programming language. Examples of such languages include Python, Java, C++, and Javascript.

The essential property of an imperative language is that it is *procedural*. Programs in these languages describe step-by-step *procedures*, in the form of sequences of *commands*, for solving given problem instances. Commands in turn operate (1) by reading, computing with, and updating values stored in a *memory*, and (2) by interacting with the world outside of the computer by executing input and output (I/O) commands.

Input (or *read*) commands obtain data from *sensors.* Sensors include mundane devices such as computer mice, track-pads, and keyboards. They also include sensors for temperature, magnetism, vibration, chemicals, biological agents, radiation, and face and license plate recognition, and much more. Sensors convert physical phenomena in the world into digital data that programs can manipulate. Computer programs can thus be made to *compute about reality beyond the computing machine*.

Output (or *write*) commands turn data back into physical phenomena in the world. The cruise control computer in a car is a good example. It periodically senses both the actual speed of the car and the desired speed set by the driver. It then computes the difference and finally finally it outputs data representing that difference to an *actuator* that changes the physical accelerator and transmission settings of the car to speed it up or slow it down. Computer programs can thus also be made to *manipulate reality beyond the computing machine*.

A special part of the world beyond of the (core of a) computer is its *memory*. A memory is to a computer like a diary or a notebook is to a person: a place to *write* information at one point in time that can then be *read* back later on. Computers use special actuators to write data to memory, and special sensors to read it back from memory when it is needed later on. Memory devices include *random access memory* (RAM), *flash memory*, *hard drives*, *magnetic tapes*, *compact* and *bluray* disks, cloud-based data storage systems such as Amazon's *S3* and *Glacier* services, and so forth.

Sequential progams describe sequences of actions involving reading of data from sensors (including from memory devices), computing with this data, and writing resulting data out to actuators (to memory devices, display screens, and physical systems controllers). Consider the simple assignment command, $x := x + 1$. It tells the computer to

first *read* in the value stored in the part of memory designated by the variable, *x, to add one to that value, and finally to *write* the result back out to the same location in memory. It's as if the person read a number from a notebook, computed a new number, and then erased the original number and replaced it with the new number. The concept of an updateable memory is at the very heart of the imperative model of computation.

2.2 Declarative Languages for Specifications

The language of formal requirements and specifications, on the other hand, is not imperative code but *declarative* logic. Expressions in such logic will state *what* properties or relationships must hold in given situation without providing a procedures that describes *how* such results are to be obtained.

To make the difference between procedural and declarative styles of description clear, consider the problem of computing the positive square root of any given non-negative number, *x*. We can *specify* the result we seek in a clear and precise logical style by saying that, for any given non-negative number *x*, we require a value, *y*, such that $y^2 = x$. Such a *y*, squared, gives *x*, and this makes *y* a square root.

We would write this mathematically as $\forall x \in \mathbb{R} \mid x >= 0, y \in \mathbb{R} | y >= 0 \wedge y^2 = x$. In English, we'd pronounce this expression as, "for any value, *x*, in the real numbers, where *x* is greater than or equal to zero, the result is a value, *y*, also in the real numbers, where *y* is greater than or equal to zero and *y* squared is equal to *x*." (The word, *where*, here is also often pronounced as *such that*. Repeat it to yourself both ways until it feels natural to translate the math into spoken English.)

Let's look at this expression with care. First, the symbol, $\forall$, is read as *for all* or *for any*. Second, the symbol $\mathbb{R}$, is used in mathematical writing to denote the set of the *real numbers*, which includes the *integers* (whole numbers, such as *-1*, *0*, and *2*), the rational numbers (such as $2/3$ and *1.5*), and the irrational numbers (such as *pi* and *e*). The symbol, $\in$, pronounced as *in*, represents membership of a value, here *x*, in a given set. The expression, $\forall x \in \mathbb{R}$ thus means "for any value, *x*, in the real numbers," or just "for any real number, *x*".

The vertical bar followed by the statement of the property, *x >= 0*, restricts the value being considered to one that satisfies the stated property. Here the value of *x* is restricted to being greater than or equal to zero. The formula including this constraint can thus be read as "for any non-negative real number, *x*." The set of non-negative real numbers is thus selected as the *domain* of the function that we are specifying.

The comma is our formula is a major break-point. It separates the specification of the *domain* of the function from a formula, after the comma, that specifies what value, if any, is associated with each value in the domain. You can think of the formula after the comma as the *body* of the function. Here it says, assuming that *x* is any non-negative real numner, that the associated value, sometimes called the *image* of *x* under the function, is a value, *y*, also in the real numbers (the *co-domain* of the function), such that *y* is both greater than or equal to zero equal *and* $y^2 = x$. The symbol, $\wedge$ is the logical symbol for *conjunction*, which is the operation that composes two smaller propositions or properties into a larger one that is true or satisfied if and only if both constituent propositions or properties are. The formula to the right of the comma thus picks out exactly the positive (or more accurate a non-negative) square root of *x*.

We thus have a precise specification of the positive square root function for non-negative real numbers. It is defined for every value in the domain insofar as every non-negative real number has a positive square root. It is also a *function* in that there is *at most one* value for any given argument. If we had left out the non-negativity *constraint* on *y* then for every *x* (except *0*) there would be *two* square roots, one positive and one negative. We would then no longer have a *function*, but rather a *relation*. A function must be *single-valued*, with at most one "result" for any given "argument".

We now have a *declarative specification* of the desired relationship between *x* and *y*. The definition is clear (once you understand the notation), it's concise, it's precise. Unfortunately, it isn't what we call *effective*. It doesn't give us a way to actually *compute* the value of the square root of any *x*. You can't run a specification in the language of mathematical logic (at least not in a practical way).

2.3 Refining Declarative Specifications into Imperative Implementations

The solution is to *refine* our declarative specification, written in the language of mathematical logic, into a computer program, written in an imperative language: one that computes *exactly* the function we have specified. To refine means to add detail while also preserving the essential properties of the original. The details to be added are the procedural steps required to compute the function. The essence to be preserved is the value of the function at each point in its domain.

In short, we need a step-by-step procedure, in an imperative language, that, when *evaluated with a given actual parameter value*, computes exactly the specified value. Here's a program that *almost* does the trick. Written in the imperative language, Python, it uses Newton's method to compute *floating point* approximations of positive square roots of given non-negative *floating point* arguments.

```
def sqrt(x):
    """for x>=0, return non-negative y such that y^2 = x"""
    estimate = x/2.0
    while True:
        newestimate = ((estimate+(x/estimate))/2.0)
        if newestimate == estimate:
            break
        estimate = newestimate
    return estimate
```

This procedure initializes and then repeatedly updates the values stored at two locations in memory, referred to by the two variables, *estimate* and *newestimate*. It repeats the update process until the process *converges* on the answer, which occurs when the values of the two variables become equal. The answer is then returned to the caller of this procedure.

Note that, following good programming style, we included an English rendering of the specification as a document string in the second line of the program. There are however several problems using English or other natural language comments to document specifications. First, natural language is prone to ambiguity, inconsistency, imprecision, and incompleteness. Second, because the document string is just a comment, there's no way for the compiler to check consistency between the code and this specification. Third, in practice, code evolves (is changed over time), and developers often forget, or neglect, to update comments, so, even if an implementation is initially consistent with a such a comment, inconsistencies can and often do develop over time.

In this case there is, in fact, a real, potentially catastrophic, mathematical inconsistency between the specification and what the program computes. The problem is that in Python, as in many everyday programming languages, so-called *real* numbers are not exactly the same as the real (*mathematical*) reals!

You can easily see what the problem is by using our procedure to compute the square root of 2.0 and by then multiplying that number by itself. The result of the computation is the number *1.41421356237*, which we already know has to be wrong to some degree, as the square root of two is an *irrational* number that cannot be represented by any non-terminating, non-repeating decimal. Indeed, if we multiply this number by itself, we get the number, *1.99999999999*. We end up in a situation in which *sqrt(2.0) * sqrt(2.0)* isn't equal to 2.0!

The problem is that in Python, as in most industrial programming languages, *so-called* real numbers (often called *floating point* numbers) are represented in just 64 binary digits, and that permits only a finite number of digits after the decimal to be represented. And additional *low-order* bits are simply dropped, leading to what we call *floating-point roundoff errors.* That's what we're seeing here.

In fact, there are problems not only with irrational numbers but with rational numbers with repeating decimal expansions when represented in the binary notation of the IEEE-754 (2008) standard for floating point arithmetic. Try adding *1/10* to itself *10* times in Python. You will be surprised by the result. *1/10* is rational but its decimal form is repeating in base-2 arithmetic, so there's no way to represent *1/10* precisely as a floating point number in Python, Java, or in many other such languages.

There are two possible solutions to this problem. First, we could change the specification to require only that *y* squared be very close to *x* (within some specified margin of error). The we could show that the code satisfies this approximate definition of square root. An alternative would be to restrict our programming language to represent real numbers as rational numbers, use arbitrarily large integer values for numerators and denominators, and avoid defining any functions that produce irrational values as results. We'd represent *1/10* not as a 64-bit floating point number, for example, but simply as the pair of integers *(1,10)*.

This is the solution that Dafny uses. So-called real numbers in Dafny behave not like *finite-precision floating point numbers that are only approximate* in general, but like the *mathematical* real numbers they represent. The limitation is that not all reals can be represented (as values of the *real* type in Dafny. In particular, irrational numbers cannot be represented exactly as real numbers. (Of course they can't be represented exactly by IEEE-754 floating point numbers, either.) If you want to learn (a lot) more about floating point, or so-called *real*, numbers in most programming languages, read the paper by David Goldberg entitled, *What Every Computer Scientist Should Know About Floating-Point Arithmetic.* It was published in the March, 1991 issue of Computing Surveys. You can find it online.

2.4 Why Not a Single Language for Programming and Specification?

The dichotomy between specification logic and implementation code raises an important question? Why not just design a single language that's good for both?

The answer is that there are fundamental tradeoffs in language design. One of the most important is a tradeoff between *expressiveness*, on one hand, and *efficient execution*, on the other.

What we see in our square root example is that mathematical logic is highly *expressive*. Logic language can be used so say clearly *what* we want. On the other hand, it's hard using logic to say *how* to get it. In practice, mathematical logic is clear but can't be *run* with the efficiency required in practice.

On the other hand, imperative code states *how* a computation is to be carried out, but generally doesn't make clear *what* it computes. One would be hard-pressed, based on a quick look at the Python code above, for example, to explain *what* it does (but for the comment, which is really not part of the code).

We end up having to express *what* we want and *how* to get it in two different languages. This situation creates a difficult new problem: to verify that a program written in an imperative language satisfies, or *refines*, a specification written in a declarative language. How do we know, *for sure*, that a program computes exactly the function specified in mathematical logic?

This is the problem of program *verification*. We can *test* a program to see if it produces the specified outputs for *some* elements of the input domain, but in general it's infeasible to test *all* inputs. So how can we know that we have *built a program* right, where right is defined precisely by a formal (mathematical logic) specification) that requires that a program work correctly for all (∀) inputs?

CHAPTER

THREE

3. PROBLEMS WITH IMPERATIVE CODE

There's no free lunch: One can have the expressiveness of mathematical logic, useful for specification, or one can have the ability to run code efficiently, along with indispensable ability to interact with an external environment provided by imperative code, but one can not have all of this at once at once.

A few additional comments about expressiveness are in order here. When we say that imperative programming languages are not as expressive as mathematical logic, what we mean is not ony that the code itself is not very explicit about what it computes. It's also that it is profoundly hard to fully comprehend what imperative code will do when run, in large part due precisely to the things that make imperative code efficient: in particular to the notion of a mutable memory.

One major problem is that when code in one part of a complex program updates a variable (the *state* of the program), another part of the code, far removed from the first, that might not run until much later, can read the value of that very same variable and thus be affected by actions taken much earlier by code far away in the program text. When programs grow to thousands or millions of lines of code (e.g., as in the cases of the Toyota unintended acceleration accident that we read about), it can be incredibly hard to understand just how different and seemingly unrelated parts of a system will interact.

As a special case, one execution of a procedure can even affect later executions of the same procedure. In pure mathematics, evaluating the sum of two and two *always* gives four; but if a procedure written in Python updates a *global* variable and then incoporates its value into the result the next time the procedure is called, then the procedure could easily return a different result each time it is called even if the argument values are the same. The human mind is simply not powerful enough to see what can happen when computations distant in time and in space (in the sense of being separated in the code) interact with each other.

A related problem occurs in imperative programs when two different variables, say x and y, refer to the same memory location. When such *aliasing* occurs, updating the value of x will also change the value of y, even though no explicit assignment to y was made. A piece of code that assumes that y doesn't change unless a change is made explicitly might fail catastrophically under such circumstances. Aliasing poses severe problems for both human understanding and also machine analysis of code written in imperative languages.

Imperative code is thus potentially *unsafe* in the sense that it can not only be very hard to fully understand what it's going to do, but it can also have effects on the world, e.g., by producing output directing some machine to launch a missile, fire up a nuclear reactor, steer a commercial aircraft, etc.

CHAPTER

FOUR

4. PURE FUNCTIONAL PROGRAMMING AS RUNNABLE MATHEMATICS

What we'd really like would be a language that gives us everything: the expressiveness and the *safety* of mathematical logic (there's no concept of a memory in logic, and thus no possibility for unexpected interactions through or aliasing of memory), with the efficiency and interactivity of imperative code. Sadly, there is no such language.

Fortunately, there is an important point in the space between these extremes: in what we call *pure functional,* as opposed to imperative, *programming* languages. Pure functional languages are based not on commands that update memories and perform I/O, but on the definition of functions and their application to data values. The expressiveness of such languages is high, in that code often directly refects the mathematical definitions of functions. And because there is no notion of an updateable (mutable) memory, aliasing and interactions between far-flung parts of programs through *global variables* simply cannot happen. Furthermore, one cannot perform I/O in such languages. These languages thus provide far greater safety guarantees than imperative languages. Finally, unlike mathematical logic, code in functional languages can be run with reasonable efficiency, though often not with the same efficiency as in, say, C++.

In this chapter, you will see how functional languages allow one to implement runnable programs that closely mirror the mathematical definitions of the functions that they implement.

4.1 The identify function (for integers)

An *identity function* is a function whose values is simply the value of the argument to which it is applied. For example, the identify function applied to an integer value, *x*, just evaluates to the value of *x*, itself. In the language of mathematical logic, the definition of the function would be written like this.

$$\forall x \in \mathbb{Z}, x.$$

In English, this would be pronounced, "for all ($\forall$) values, *x*, in ($\in$) the set of integers ($\mathbb{Z}$), the function simply reduces to value of *x*, itself. The infinite set of integers is usually denoted in mathematical writing by a script or bold Z. We will use that convention in these notes.

While such a mathematical definition is not "runnable", we can *implement* it as a runnable program in pure functional language. The code will then closely reflects the abstract mathematical definition. And it will run! Here's an implementation of *id* written in the functional sub-language of Dafny.

```
function method id (x: int): int { x }
```

The code declares *id* to be what Dafny calls a "function method", which indicates two things. First, the *function* keyword states that the code will be written in a pure functional, not in an imperative, style. Second, the *method* keyword instructs the compiler to produce runnable code for this function.

Let's look at the code in detail. First, the name of the function is defined to be *id*. Second, the function is defined to take just one argument, *x*, declared of type *int*. The is the Dafny type whose values represent integers (negative, zero,

and positive whole number) of any size. The Dafny type *int* thus represents (or *implements*) the mathematical set, $\mathbb{Z}$, of all integers. The *int* after the argument list and colon then indicates that, when applied to an int, the function returns (or *reduces to*) a value of type *int*. Finally, within the curly braces, the expression *x*, which we call the *body* of this function definition, specifies the value that this function reduces to when applied to any *int*. In particular, when applied to avalue, *x*, the function application simply reduces to the value of *x* itself.

Compare the code with the abstract mathematical definition and you will see that but for details, they are basicaly *isomorphic* (a word that means identical in structure). It's not too much of a stretch to say that pure functional programs are basically runnable mathematics.

Finally, we need to know how expressions involving applications of this function to arguments are evaluated. They fundamental notion at the heart of functional programming is this: to evaluate a function application expression, such as *id(4)*, you substiute the value of the argument (here *4*) for every occurence of the argument variable (here *x*) in the body of the function definition, the you evaluate that expression and return the result. In this case, we substite *4* for the *x* in the body, yielding the literal expression, *4*, which, when evaluated, yeilds the value *4*, and that's the result.

4.2 Data and function types

Before moving on to more interesting functions, we must mention the concepts of *types* and *values* as they pertain to both *data* and *functions*. Two types appear in the example of the *id* function. The first, obvious, one is the type *int*. The *values* of this type are *data* values, namely values representing integers. The second type, which is less visible in the example, is the type of the the function, *id*, itself. As the function takes an argument of type *int* and also returns a value of type *int*, we say that the type of *id* is $int \rightarrow int$. You can pronounce this type as *int to int*.

4.3 Other function values of the same type

There are many (indeed an uncountable infinity of) functions that convert integer values to other integer values. All such functions have the same type, namely $int \rightarrow int$, but they constitute different function *values*. While the type of a function is specified in the declaration of the function argument and return types, a function *value* is defined by the expression comprising the *body* of the function.

An example of a different function of the same type is what we will call *inc*, short for *increment*. When applied to an integer value, it reduces to (or *returns*) that value plus one. Mathematically, it is defined as $\forall x \in \mathbb{Z}, x + 1$. For example, *inc(2)* reduces to *3*, and *inc(-2)*, to *-1*.

Here's a Dafny functional program that implements this function. You should be able to understand this program with ease. Once again, take a moment to see the relationship between the abstract mathematical definition and the concrete code. They are basically isomorphic. The pure functional programmer is writing *runnable mathematics*.

```
function method inc (x: int): int { x + 1 }
```

Another example of a function of the same type is, *square*, defined as returing the square of its integer argument. Mathematically it is the function, $\forall x \in \mathbb{Z}, x * x$. And here is a Dafny implementation.

```
function method h (x: int): int { x * x }
```

Evaluating expressions in which this function is applied to an argument happens as previously described. To evaluate *square(4)*, for example, you rewrite the body, *x * x*, replacing every *x* with a *4*, yielding the expression *4 * 4*, then you evaluate that expression and return the result, here *16*. Function evaluation is done by substituting actual parameter values for all occurrences of corresponding formal parameters in the body of a function, evaluating the resulting expression, and returning that result.

Recursive function definitions and implementations ==========+

Many mathematical functions are defined *recursively*. Consider the familiar *factorial* function. An informal explanation of what the function produces when applied to a natural number (a non-negative integer), *n*, is the product of natural numbers from *1* to *n*.

That's a perfectly understandable definition, but it's not quite precise (or even correct) enough for a mathematician. There are at least two problems with this definition. First, it does not define the value of the function *for all* natural numbers. In particular, it does not say what the value of the function is for zero. Second, you can't just extend the definition by saying that it yields the product of all the natural numbers from zero to *n*, because that is always zero!

Rather, if the function is to be defined for an argument of zero, as we require, then we had better define it to have the value one when the argument is zero, to preserve the product of all the other numbers larger than zero that we might have multiplied together to produce the result. The trick is to write a mathematical definition of factorial in two cases: one for the value zero, and one for any other number.

$$factorial(n) := \forall n \in \mathbb{Z} \mid n >= 0, \begin{cases} \text{if n=0,} & 1, \\ \text{otherwise,} & n * factorial(n-1). \end{cases}$$

To pronounce this mathematical definition in English, one would say that for any integer, *n*, such that *n* is greater than or equal to zero, *factorial(n)* is one if *n* is zero and is otherwise *n* times *factorial(n-1)*.

Let's analyze this definition. First, whereas in earlier examples we left mathematical definitions anonymous, here we have given a name, *factorial*, to the function, as part of its mathematical definition. We have to do this because we need to refer to the function within its own definition. When a definition refers to the thing that is being defined, we call the definition *recursive.*

Second, we have restricted the *domain* of the function, which is to say the set of values for which it is defined, to the non-negative integers only, the set known as the *natural numbers*. The function simply isn't defined for negative numbers. Mathematicians usually use the symbol, $\mathbb{N}$ for this set. We could have written the definition a little more concisely using this notation, like this:

$$factorial(n) := \forall n \in \mathbb{N}, \begin{cases} \text{if n=0,} & 1, \\ \text{otherwise,} & n * factorial(n-1). \end{cases}$$

Here, then, is a Dafny implementation of the factorial function.

```
function method fact(n: int): int
   requires n >= 0 // for recursion to be well founded
{
    if (n==0) then 1
    else n * fact(n-1)
}
```

This code exactly mirrors our first mathematical definition. The restriction on the domain is expressed in the *requires* clause of the program. This clause is not runnable code. It's a specification: a *predicate* (a proposition with a parameter) that must hold for the program to be used. Dafny will insist that this function only ever be applied to values of *n* that have the *property* of being $>= 0$. A predicate that must be true for a program to be run is called a *pre-condition*.

To see how the recursion works, consider the application of *factorial* to the natural number, *3*. We know that the answer should be *6. The evaluation of the expression, *factorial(3)*, works as for any function application expression: first you subsitute the value of the argument(s) for each occurrence of the formal parameters in the body of the function; then you evaluate the resulting expression (recursively!) and return the result. For *factorial(3)*, this process leads through a

sequence of intermediate expressions as follows (leaving out a few details that should be easy to infer):

$$
\begin{aligned}
& factorial\ (3) && \text{; a function application expression} \\
& if\ (3 == 0)\ then\ 1\ else\ (3 * factorial\ (3-1)) && \text{; expand body with parameter/argument substitution} \\
& if\ (3 == 0)\ then\ 1\ else\ (3 * factorial\ (2)) && \text{; evaluate } (3-1) \\
& if\ false\ then\ 1\ else\ (3 * factorial\ (2)) && \text{; evaluate } (3 == 0) \\
& (3 * factorial\ (2)) && \text{; evaluate } ifThenElse \\
& (3 * (if\ (2 == 0)\ then\ 1\ else\ (2 * factorial\ (1)))) && \text{; etc} \\
& (3 * (2 * factorial\ (1))) \\
& (3 * (2 * (if\ (1 == 0)\ then\ 1\ else\ (1 * factorial\ (0))))) \\
& (3 * (2 * (1 * factorial\ (0)))) \\
& (3 * (2 * (1 * (if\ (0 == 0)\ then\ 1\ else\ (0 * factorial\ (-1)))))) \\
& (3 * (2 * (1 * (if\ true\ then\ 1\ else\ (0 * factorial\ (-1)))))) \\
& (3 * (2 * (1 * 1))) \\
& (3 * (2 * 1)) \\
& (3 * 2) \\
& 6
\end{aligned}
$$

The evaluation process continues until the function application expression is reduced to a data value. That's the answer!

It's important to understand how recursive function application expressions are evaluated. Study this example with care. Once you're sure you see what's going on, go back and look at the mathematical definition, and convince yourself that you can understand it *without* having to think about *unrolling* of the recursion as we just did.

Finally we note that the the precondition is essential. If it were not there in the mathematical definition, the definition would not be what mathematicians call *well founded*: the recursive definition might never stop looping back on itself. Just think about what would happen if you could apply the function to *-1*. The definition would involve the function applied to *-2*. And the definition of that would involve the function applied to *-3*. You can see that there will be an infinite regress.

Similarly, if Dafny would allow the function to be applied to *any* value of type *int*, it would be possible, in particular, to apply the function to negative values, and that would be bad! Evaluating the expression, *factorial(-1)* would involve the recursive evaluation of the expression, *factorial(-2)*, and you can see that the evaluation process would never end. The program would go into an "infinite loop" (technically an unbounded recursion). By doing so, the program would also violate the fundamental promise made by its type: that for *any* integer-valued argument, an integer result will be produced. That can not happen if the evaluation process never returns a result. We see the precondition in the code, implementing the domain restriction in the mathematical definition, is indispensible. It makes the definition sound and it makes the code correct!

4.4 Dafny is a Program Verifier

Restricting the domain of factorial to non-negative integers is critical. Combining the non-negative property of ever value to which the function is applied with the fact that every recursive application is to a smaller value of *n*, allows us to conclude that no *infinite decreasing chains* are possible. Any application of the function to a non-negative integer *n* will terminate after exactly *n* recursive calls to the function. Every non-negative integer, *n* is finite. So every call to the function will terminate.

Termination is a critical *property* of programs. The proposition that our factorial program with the precondition in place always terminates is true as we've argued. Without the precondition, the proposition is false.

Underneath Dafny's "hood," it has a system for proving propositions about (i.e., properties of) programs. Here we see that It generates a propostion that each recursive function terminates; and it requires a proof that each such proposition is true.

With the precondition in place, there not only is a proof, but Dafny can find it on its own. If you remove the precondition, Dafny won't be able to find a proof, because, as we just saw, there isn't one: the proposition that evaluation of the function always terminates is not true. In this case, because it can't prove termination, Dafny will issue an error stating, in effect, that there is the possibility that the program will infinitely loop. Try it in Dafny. You will see.

In some cases there will be proofs of important propositions that Dafny nevertheless can't find it on its own. In such cases, you may have to help it by giving it some additional propositions that it can verify and that help point it in the right direction. We'll see more of this later.

The Dafny language and verification system is powerful mechansim for finding subtle bugs in code, but it require a knowledge of more than just programming. It requires an understanding of specification, and of the languages of logic and proofs in which specifications of code are expressed and verified.

CHAPTER

FIVE

5. FORMAL VERIFICATION OF IMPERATIVE PROGRAMS

In this chapter, we first elaborate on the idea that pure functional programming make for mathematically clear but potentially inefficient specifications, while imperative code makes for efficient code but is hardly clear as to its purpose, and is thus hard to reason about. To get the benefits of both, we use functional programming to write key parts of specifications for imperative code, and then we use tools or manual methods to *prove* that the imperative code does what such a speification requires.

5.1 Performance vs. Understandability

To get a clearer sense of the potential differences in performance between a pure functional program and an imperative program that compute the same function, and tradeoffs one makes between clarity of intent and execution speed, consider our recursive definition of the Fibonacci function.

We start off knowing that if the argument to the function, *n*, is *0* or *1*, the value of the function for that *n* is just *n* itself. In other words, the sequence, *fib(i)* of *Fibonacci numbers indexed by i*, starts with, $[0, 1, \ldots]$. For any *n >= 2*, *fib(n)*, is the sum of the previous two values. To compute the *n'th (n >= 2)* Fibonacci number, we can thus start with the first two, sum them up to get the next one, then iterate this process, computing the next value on each iteration, until we've got the result.

Footnote: by convention we index sequences starting at zero rather than one. The first element in such a sequence thus has index *0*, the second has index *1*, and the *n'th* has index *n - 1*. For example, *fib(6)* refers to the *7th* Fibonacci number. You should get used to thinking in terms of zero-indexed sequences.

Now consider our recursive definition, *fib(n)*. It's *pure math*: concise, precise, elegant. And because we've written it in a functional language, we can even run it. However, it might not give us the performance we require. An imperative program, by constrast, is *code*. It's cryptic but it can be very efficient when run.

To get a sense of performance differences, consider the evaluation of each of two programs to compute *fib(5)*: our functional program and an imperative one that we will develop in this chapter.

Consider the imperative program. If the argument, *n*, is either zero or one, the answer is just returned. If *n >= 2* an answer has to be computed. In this case, the program will repeatedly add together the previous two values of the function, starting with *0* and *1*, until it computes the result for *n*. The program returns that value.

For a given value of *n*, what is the cost of computing an answer? The cost will be dominated by the work done inside the loop body; and on each iteration of the loop, a fixed amount of work is done; so it's not a bad idea to use the number of loop body executions as a measure of the cost of computing an answer for an argument, *n*.

So, what does it cost to compute *fib(5)*? Well, we need to execute the loop body to compute *fib(i)* for values of *i* of *2, 3, 4,* and *5*. It thus takes *4* loop body iterations to compute *fib(5)*. To compute the 10th element requires that the loop body execute for *i* in the range of *[2, 3, ..., 10]*. That's nine iterations. It's easy to see that for any value of *n*, the cost to compute *fib(n)* will be *n-1* loop body iterations. We can compute the *100,000th* Fibonacci number by running a simple loop body *about* that many times. On a modern computer, the computation will be completed very quickly.

The functional program, on the other hand, is evaluated by repeated evaluation of nested recursive function applications until base cases are reached. Let's think about the cost of evaluation for increasing values of *n* and try to see a pattern. We'll measure computational complexity now in terms of the number of function evaluations (rather than loop bodies executed) required to produce a final answer.

To compute *fib(0)* or *fib(1)* requires just *1* function evaluation (the first and only call to the function), as these are base cases requiring no further recursion. To compute *fib(2)* however requires *3* evalations of *fib*: one for each of *fib(1)* and *fib(0)* plus the evaluation of the top-level function. The relationship between *n* and the number of function evaluations currently looks like this: $\{(0,1),(1,1),(2,3),...\}$. The first element of each pair is *n* and the second element is the cost to compute *fib(n)*.

What about when *n* is *3*? Computing this requires answers for *fib(2)*, which by the resuts we just computed costs *3* evaluations, and for *fib(1)*, which costs *1*, for a total of *5* evaluations including the top-level evaluation. Computing *fib(4)* requires that we compute *fib(3)* and *fib(2)*, costing *5 + 3*, or *8* evaluations, plus the original, top-level call, for a total of 9. For *fib(5)* we need *9 + 5*, or *14* plus one more, making *15* evaluations. The relation of cost to *n* (the problem size) is now like this: $\{(0,1),(1,1),(2,3),(3,5),(4,9),(5,15),...\}$.

In general, the number of evaluations needed to evaluate *fib(i+1)* is the sum of the numbers required to evaluate *fib(i)* plus the number to evaluate *fib(i-1)* plus *1*. If we use *C* to represent the cost function, then we could say, $C(n) = C(n-1) + C(n-2) + 1$. This kind of function is called a recurrence relation, and there are clever ways to solve such functions to determine what function *C* may be. Of course we can also write a recursive function to compute *C(n)*, if we need only to compute it for relatively small values of *n*.

Now that we have the formula, we can quickly compute the costs to compute *fib(n)* for numerous values of *n*. The number of evaluations needed to compute *fib(6)* is *15 + 9 + 1*, i.e., 25. For *fib(7)* it's *41*. For *fib(8), *67*; for *fib(9), 109*; for *fib(10), 177*; and for *fib(11), 286* function evaluations.

One thing is clear: The cost to compute the *n'th* Fibonacci number, as measured by the number of function evaluations, using our beautiful functional program, is growing much more quickly than *n* itself, and indeed it is growing faster and faster as *n* increases. We would say the cost is *super-linear*, whereas with our imperative program, the number of loop body interations grows *linearly* in *n*.

How exactly does the cost of the pure functional program compare? One thing to notice is that the cost of computing a Fibonacci element with our functional program is related to the Fibonacci sequence itself! The first two values in the *cost* sequence are *1* and *1*, and each subsequence element is the sum of the previous two *plus 1*. It's not exactly the Fibonacci sequence, but it turns out to grow at a very similar rate. The Fibonacci sequence, thus also the cost of computing it recursively, grows at what turns out to be a rate *exponential* in *n*, with an exponent of about *1.6*. Increasing *n* by *1* doesn't just add a little to the cost; it almost doubles it (multiplying it by a factor of *1.6*).

No matter how small the exponent (with any exponent greater than one), exponential functions eventually grow very quickly. In the limit, any exponential function grows faster than any polynomial no matter how high in rank it is and no matter how large its coefficients are.

The exponential-in-*n* cost of our clear but inefficient functional program grows far faster than the cost of our ugly but efficient imperative program as we increase *n*. For any even modestly large value of *n* (e.g., greater than 50 or so), it will be impractical to use the pure functional program, whereas the imperative program will reasonably run quickly even on a small personal computer for values of *n* well into in the millions. What eventually slows it down is not the number of additions that it has to do but the sizes of the numbers that it has to add.

You can already see that the cost to compute *fib(n)* recursively for values of *n* larger than just ten or so is much greater than the cost to compute it iteratively. Our mathematical/functional definition is clear ("intellectually tractable") but inefficient. The imperative program, on the other hand, is efficient, but not at all transparent. We need the latter program for practical computation. But how do we ensure that it implements the same function that we expressed in our elegant mathematical definition?

5.2 Specification, Implementation, and Verification

We address such problems by combining a few ideas. First, we use logic, including mathematical specifications written in part using functional programming, to express *declarative* specifications. Such specification precisely define *what* a given imperative program must compute, and in particular what results it must return as a function of the arguements it receives, without saying *how* the computation should be done.

We can use functions defined in the pure functional programming style as parts of specifications, e.g., as giving a mathematical definition of the *factorial* function that an imperative program will then have to implement.

Second, we implement the specified program in an imperative language. Ideally we do so in a way that supports logical reasoning about its behavior. For example, we have to specify not only the relationship between argument and result values that are required, but also how loops are designed to work in our code. We then need to design loops in ways that make it easier to explain, in formal logic, how they do what they are meant to do.

Finally, we use logical proofs to *verify* that the program satisifies its specification. Later in this course, we'll see how to create such proofs ourselves. For now we'll be happy to let Dafny generate them for us mostly automatically!

The rest of this chapter develops these ideas in more depth with concrete examples. First we explain how formal specifications in mathematical logic for imperative programs are often organized. Next we explore how writing imperative programs without the benefits of specification languages and verifications tools can make it hard to spot bugs in code. Next we enhance our implementation of the factorial function with specifications, show how Dafny flags the bug, and fix the program. Doing this requires that we deepen the way we understand loops. We end with a detailed presentation of the verification of an imperative program to compute values in the Fibonacci sequence. Given any natural number *n*, our program must return the value of *fib(n)*, but it must also do it efficiently. The design and precise, logical description of key properties of a loop is once again the heart of the problem. We will see how Dafny can help us to reason rigorously about loops, and that giving it a little help enables it to reason about them for us.

5.3 Declarative Input-Output Specifications

First, we use mathematical logic to *declaratively specify* properties of the behaviors that we require of programs written in *imperative* languages. For example, we might require that, when given *any* natural number, *n*, a program compute and return the value of the *factorial* of *n*, the mathematical definition of which we've given as *fact(n)*. In general, we want to specify how the results returned by an imperative program relate to the arguments on which it was run. We call such a specification an *input-output* specification. (Here we ignore *side-effect* behaviors such as reading from and writing to input and output devices.)

Specifications about required relationships between argument values and return results specify *what* a program must compute without specifying how it should be done. Specifications are thus *abstract*: they omit *implementation details*, leaving it to the programmer to decide how best to *refine* the specification into efficient code.

For example we might specify that a program (1) must accept any integer valued argument greater than or equal to zero (a piece of a specification that we call a *precondition*), and (2) that as long as the precondition holds, then it must return the factorial of the given argument value (a *postcondition*).

5.3.1 Input-Output Relations

In purely mathematical terms, a specification of this kind defines a *binary relation* between argument (input) and return (output) values, and imposes on the program a requirement that whenever it is given the first value in such an *input-output* pair, it must compute a second (output) value so that the pair, $(input, output)$, is in the specified relation.

5.3.2 Relations and Functions

A binary relation in ordinary mathematics is just a set of pairs of values. A function is a binary relation with at most one pair with a given first value. A function is a *single-valued* relation. What we often need to specify, in particular, is an input-output *function*.

For example, pairs in the factorial relation include $(0, 1), (1, 1), (2, 2), (3, 6), (4, 24)$ and $(5, 120)$, but not the pair $(5, 25)$. Some of the pairs in the Fibonacci relation include $\{(0, 0), (1, 1), (2, 1), (3, 2), (5, 3)$ and *(6,5)*. These relations are also *functions* because there is *at most* one pair with a given first element. Finally, these functions are also said to be *total* because for *every* natural number, there is a pair with that number as its first element.

On the other hand, square root is a *relation*, a set of pairs of real numbers, but not a *function*, because it is not singled valued. Both of the pairs, $(4, 2)$ and $(4, -2)$, which are distinct but have same first element, are in the relation. That is so because both *2* and *-2* are square roots of *4*.

5.3.3 Total and Partial Functions

We also note that the square root relation *on the real numbers* is what we call *partial* rather than total: in that it is not defined for some real numbers. In particular, it is not defined for (i.e., it does not have any pairs where the first element is) any negative real number.

5.3.4 Turning Partial Functions into Total Functions

Partial functions and non-function relations both present problems for programmers. Let's first consider relations that sometimes have *more* than one value of a given type for a given argument. What value should a program return?

The square root function is a good example. Given a positive argument there will be *two* square roots, one positive and one negative. If the function is require to return a single number as an answer, which one should it return?

There is really no good answer. Rather, the solution is usually to change the program specification slightly. For example, rather than promising to return *the* square root (a concept that is not well defined when there are two square roots for the same number) such a program might promise to return the non-negative square root, of which there is always just one (given a non-negative argument. What we have done here is to implement a different relation, and one that is now also a function.

A different way to re-formulate the square root *relation* as a *function* would be to view it as returning a single *set* of values as a result: a set containing all of the square roots of a given argument. The pair $(4, \{2, -2\})$ is in this relation, for example, and the relation is also a function in that there is only one such pair with any given first element.

So far we have dealt with the situation where a relation holds more than one result for a given argument. The other difficult situation occurs when there is no result or a given argument, i.e., when the function or relation is undefined for some argument values. What should a program return then?

Once again, there's no good answer. Rather, we generally tweak the specification to require the implementation of a slightly different relation. One approach would be to narrow the domain of values that the *program* can take to the domain on which the actual mathematical function is defined. So instead of specifying a square root function as taking any real number, we could speficy that it requires that an argument value be non-negative. When we add such a precondition to a method or function specification in Dafny, the effect is that Dafny checks every place in the code where the method or function is called to verify that the argument values satisfy that pre-condition.

Alternately, we might "tweak" the type of the return value, so that the program can return some value of the promised type, even if the underlying mathematical function is not defined for the arguments. So, for example, if instead of promising to return a single number as a square root we promise to return a set of numbers, then in cases where the function is undefined, we just return the empty set of numbers. In this case, the empty set as a return value can be interpreted as signifying that no numerical answer could be returned.

Finally, in languages such as Java and Python, when a program encounters a state where a valid value cannot be computed and returned, it can invoke an error handling routine to take some kind of "exceptional" action. This is the purpose of exceptions in Java, Python, etc. We will not entertain the use of exceptions in this course.

5.4 Imperative Implementation

Having written a formal specification of the required *input-output* behavior of a program, we next write imperative code in a manner, and in a language, that supports the use of formal logic to *reason* about whether the program refines (implements) its formal specification. One can use formal specifications when programming in any language, but it helps greatly if the language has strong, static type checking. It is even better if the language supports formal specification and logical reasoning mechanisms right alongside of its imperative and functional programming capabilities. Dafny is such a language and system. It is not just a language, but a verifier.

In addition to choosing a language with features that help to support formal reasoning, we sometimes also aim to write imperative code in a way that makes it easier to reason about formally. As we'll see below, for example, the way that we write our while loops can make it easier or harder to reason about their correctness. Even whether we iterate from zero up to *n* or from *n* down to zero can affect the difficulty of writing specification elements for a program.

5.5 Formal Verification

The aim of formal verification is to deduce (to use deductive logic to *prove*) that, as written, a program satisfies its specification. In more detail, if we're given a program, *C* with a precondition, *P*, and a postcondition *Q*, we want a proof that verifies that if *C* is started in a state that satisfies *P* and if it terminates (doesn't go into an infinite loop), that it ends in a state that satisfies *Q*. We call this property *partial correctness*.

We write the proposition that *C* is partially correct (that if it's started in a state that satisfies the assertion, *P*, and that if it terminates, then it will do so in a state that satisfies assertion *Q*) as $P\{C\}Q$. This is a so-called *Hoare triple*, named after the famous computer scientist, Sir Anthony (Tony) Hoare. It is nothing other than a proposition that claims that *C* satisfies its *pre-condition/post-condition* specification. Another way to read it is as saying that the combination of the pre-condition being satisfied and the the program being run implies that the post-condition will be satisfied.

In addition to a proof of partial correctness, we usually do want to know that a program also does always terminate. When we have a proof of both $P\{C\}Q$ and that the program always terminates, then we have a proof of *total correctness*. Dafny is a programming system that allows us to specify *P* and *Q* and it then formally, and to a considerable extent automatically, verifies both *P { C } Q* and termination. That is, Dafny produces proofs of total correctness.

It is important to bear in mind that a proof that a program refines (implements) its formal specification does not necessarily mean that it is fit for its intended purpose! If the specification is wrong, then all bets are off, even if the program is correct relative to its specification. The problem of *validating* specification againts real-world needs is separate from that of *verifying* that a given program implements its specification correctly. Formal methods can help here, as well, by verifying that *specifications* have certain desired properties, but formal validation of specifications is not our main concern at the moment.

5.6 Case Study: The Factorial Function

So far the material in this chapter has been pretty abstract. Now we'll see what it means in practice.

5.6.1 A Buggy Implementation

To start, let's consider an ordinary imperative program, as you might have written in Python or Java, for computing values of the factorial function. The name of the function is the only indication here of the intended behavior of this program. There is no clear specification.

The program takes an argument of type nat (which guarantees that the argument has the property of being non-negative). It then returns a nat which the programmer implicitly claims (given the function name) is the factorial of the argument.

```
method factorial(n: nat) returns (f: nat)
{
    if (n == 0)
    {
        return 1;
    }
    var t: nat := n;
    var a: nat := 1;
    while (t !=  0)
    {
        a := a * n;
        t := t - 1;
    }
    f := a;
}
```

It's not immediately obvious whether this code is correct or not, relative to what we know it's meant to do. Sadly, this program also contains a bug. Try to find it. Reason about the behavior of the program when the argument is 0, 1, 2, 3, etc. Does it always compute the right result? Where is the bug? What is wrong? And how could this logical error have been detected automatically?

5.6.2 Specifications Establish Correctness Criteria

A key problem is that the program lacks a precise specification. The program does *something*, taking a nat and possibly returning a *nat* (unless it goes into an infinite loop) but there's no way to analyze its correctness in the absence of a specification that defines what it even means to be correct.

Now let's see what happens when we add a formal specification. Look at the following code block. That *n >= 0* continues to be expressed by the *type* of the argument, *n*, being *nat*. However, we have now added a postcondition that *ensures* that the return result will be the factorial of *n* as defined by our functional program! What we assert is that the result produced by our imperative code is the same result that *would have been produced* if we had run the functional program.

```
method factorial(n: nat) returns (f: nat)
    ensures f == fact(n)
{
    if (n == 0)
    {
        return 1;
    }
    var t := n;
    var a := 1;
    while (t !=  0)
    {
        a := a * n;
        t := t - 1;
    }
```

```
    return a;
}
```

With a specification in place, Dafny now reports that it cannot guarantee—formally prove to itself—that the *postcondition* is guaranteed to hold. Generating proofs is hard, not only for people but also for machines. In fact, one of seminal results of 20th century mathematical logic was to prove that there is no general-purpose algorithm for proving propositions in mathematical logic. That's good news for mathematicians! If this weren't true, we wouldn't need them!

So, the best that a machine can do is to try to find a proof for any given proposition. Sometimes proofs are easy to generate. For example, it's easy to prove *1 = 1* by the *reflexive* propery of equality. Other propositions can be hard to prove. Proving that programs in imperative languages satisfy declarative specifications can be hard.

When Dafny fails to verify a program (find a proof that it satisfies its specification), there is one of two reasons. Either the program really does fail to satisfy its specification; or the program is good but Dafny does not have enough information to know how to prove it.

With the preceding program, the postcondition really isn't satisfied due to the bug in the program. When Dafny fails to verify, it gives us a strong reason to double-check our code to be sure we have not made some kind of mistake in reasoning.

But even if the program were correct, Dafny would still need a little more than is given here to prove it. In particular, Dafny would need a litte more information about how the while loop behaves. It turns out that providing such extra information about while loops is where much of the difficulty lies.

5.6.3 A Verified Implementation of the Factorial Function

Here, then, is a verified imperative program for computing factorial. We start by documenting the overall program specification. The key element here is the ensures clause. This clause links our imperative program with our functional specification and tells Dafny to make sure that the reuqired relationship holds.

```
method verified_factorial(n: nat) returns (f: nat)
    ensures f == fact(n)
```

Now for the body of the method. First, if we're looking at the case where *n==0* we just return the right answer immediately. There is no need for any further computation.

```
if (n == 0)
{
    return 1;
}
```

The rest of the code handles the case where *n > 1*. At this point in the program execution, we believe that *n* must be greater than zero. We would have just returned if it were zero, and it can't be negative because its type is *nat*. We can nevertheless formally assert (write a proposition about the state of the program) that *n* is greater than zero. Dafny will try to (and here will successfully) verify that the assertion is true at this point in the program, no matter what path through conditionals, while loops, commands led to this point in the program.

```
assert n > 0;
```

To compute an answer for the non-zero input case, we will use a loop. We can do this by using a variable, a, to hold a "partial factorial value" in the form of a product of the numbers from n down to a loop index, "i," that we start at n and decrement, terminating the loop when *n==0*.

At each point just before, during, and right after the loop, *a* is a product of the numbers from *n* down to but not including *i*, and the value of *i* represents how much product-computing work remains to be done. So, for example, if

we're computing factorial(10) and a holds the value *10 * 9*, then *i* must be *8* because multiplying *a* by the factors from *8* to *1* remains to be done.

A critical "invariant" then is that if you multiply *a* by the factorial of *i* you get the the factorial of *n*. When we say that this is an invariant, we mean that it holds before and also after any execution of the loop body, but not necessarily within the loop body. In particular, when *i* gets down to *0*, this relation means that *a* must contain the final result, because *a * fact(0)* will then equal *fact(n)* and *fact(0)* is just *1*, so *a* must equal *fact(n)*.

This is how we design loops so that we can be confident that they do what we want tem to do. So now let's go through the steps required to implement our looping strategy.

Step 1. Set up state for the loop to work. We first initializie a := 1 and i := n.

```
var i: nat := n;    // nat type of i explicit
var a := 1;         // can let Dafny infer it
```

It would now be a good idea to ask Dafny to check that the invariant holds. See the next bit of code, below. Note that we are again using our pure functional definition, *fact*, as a *specification* of the function we're implementing.

In Dafny, we can use matnematical logic to express what must be true at any given point in the execution of a program in the form of an "assertion." Here we assert that our loop invariant holds. The Dafny verifier tries to prove that the assertion is a true propsition about the state of the program when control reaches this point, no matter what path might have been taken to arrive at this point.

```
assert a * fact(i) == fact(n); // "invariant"
```

Step 2: Now we write the actual loop command. Recall how a *while* loop works. To evaluate a loop, one evaluates the loop condition. If the result is false, the loop body does not execute and the loop terminates. Otherwise, the loop body is executed once and then the whole loop is run again (starting with a new evaluation of the loop condition).

We want our loop body to run at least once, as we already handled the case where it doesn't need to run at all. It will run if i > 0. What is i? We initialized it to n and haven't change it since then so it must still be equal to n. Do we know that n is greater than 0? We do, because (1) it can't be negative owning to its type, and (2) it can't be 0 because if it were 0 the program would already have returned.

We can now do better than just reasoning in our heads. We can also use logic to express what we believe to be true and let Dafny try to check it for us automatically.

```
assert i > 0;
```

Now if *i* is one, then the loop body will run once. The value of *a*, which starts at 1, will be multiplied by i, which is 1, then i will be decremented, taking the the value 0. The loop will be run again, but the loop condition will be found to be false, and to the loop body will not be executed and the loop will terminate. When it does, it will leave *a* with the value 1, which is the right answer.

```
while (i >  0)
    invariant 0 <= i <= n
    invariant fact(n) == a * fact(i)
{
    a := a * i;
    i := i - 1;
}
```

If *i* is greater than 1, the loop body will execute, multiplying *a* by the current value of *i* and *i* will be decremented. The vaue of *a* will be the partial value of the factorial computed so far, and the value of *i* will represent the work that remains to be done. When *i* reaches zero, all the work will be done, and *a* will contain the final result.

However, Dafny cannot determine on its own that this will be the case. What it needs to know to reason "mechanically" about the program is a bit of additional information about what remains true no matter *how* many times the loop body

executes (zero or more). That information is expressed in the loop *invariants*. The first one is true but not of much use. The second one is the key to enabling Dafny to verify that after the loop, *a == fact(n)*.

The invariant itself just says that at all points before and after the loop body executes, that partial factorial value computed so far times the factorial of *i* (which remains to be computed) is the answer that we seek. Once the loop is done we (and Dafny) *also* know that *i == 0*. It is the combination of the invariant and this fact that enables Dafny to see that it must be the case that *a == fact(n)*.

We can verify by using asserts after the loop that our beliefs about what the state of the program must be are correct. First, let's have Dafny check that the loop condition is now false.

```
assert !(i > 0);
```

We can also have Dafny check that our loop invariant still holds.

```
assert a * fact(i) == fact(n);
```

And now comes the most crucial step of all in our reasoning. We can deduce that *a* now holds the correct answer. That this is so follows from the conjunction of the two assertions we just made. First, that *i* is not greater than *0* and given that its type is *nat*, the only possible value it can have now is *0*. That's what we'd expect, as it is the condition on which the loop terminates (which it just did). But better than just saying all of this, let us also formalize, document, and check it using the Dafny verifier.

```
assert i == 0;
```

Now it's easy to see. No matter what value *i* has, we know that the loop invariant holds: $a * fact(i) == fact(n)$, and we also know that *i == 0*. So it must be that $a * fact(0) == fact(n)$. And fact(0) is *1* (from its mathematical definition). So it must be that *a == fact(n)*. And Dafny confirms it!

```
assert a == fact(n);
```

We thus have the answer we need to return. Dafny verifies that our program satisfies its formal specification. We no longer have to pray. We *know* that our program is right and Dafny confirms our belief.

```
return a;
```

Mathematical logic is to software as the calculus is to physics and engineering. It's not just an academic curiosity. It is a critical intellectual tool, inceasingly used for precise specification and semi-automated reasoning about and verification of real programs.

5.7 Case Study: The Fibonacci Function

Similarly, here is a verified imperative implementation of the Fibonacci function. We start by adding a specification in the form of an ensures clause, appealing to our functional program, to tell Dafny what the imperative program must compute.

```
method fibonacci(n: nat) returns (r: nat)
    ensures r == fib(n)
```

Now for the body. First we represent values for the two cases where the result requires no further computation. Initially, *fib0* will store the value of *fib(0)*, namely *0*, and *fib1* will store the value of *fib(1)*, namely *1*.

```
var fib0, fib1 := 0, 1; //parallel assignment
```

Next, we test to see if either of these cases applies, and if so we just return the appropriate result.

```
if (n == 0) { return fib0; }
if (n == 1) { return fib1; }
```

At this point, we know something more about the state of the program than was the case when we started. We can deduce that *n* has to be greater than or equal to *2*. This is because it initially had to be greater than or equal to zero due to its type, and we would already have returned if it were *0* or *1*. It must now be *2* or greater. We can assert this proposition about the state of the program at this point, and Dafny will verify it for us.

```
assert n >= 2;
```

So now we have to deal with the case where *n >= 2*. Our strategy for computing *fib(n)* in this case is, again, to use a while loop. We will establish a loop index *i*. Our design will be based on the idea that at the beginning and end of each loop iteration (we are currently at the beginning), we will have already computed *fib(i)* and that its value is in *fib1*. We've already assigned the value of *fib(0)* to *fib0*, and *fib(1)* to *fib1*, so to set up the desired state, we initialize *i* to be *1*.

```
var i := 1;
```

We can now assert and Dafny can verify a number of conditions that we expect and require to hold. First, *fib1* equals *fib(i)*. To compute the next (*i+1st*) Fibonacci number, we need not only the value of *fib(i)* but also *fib(i-1)*. We will thus also want *fib0* to hold this value at the start and end of each loop iteration. Indeed we do have this state of affairs right now.

```
assert fib1 == fib(i);
assert fib0 == fib(i-1);
```

To compute *fib(n)* for any *n* greater than or equal to *2* will require at least one execution of the loop body. We'll thus set our loop condition to be $i < n$. This ensures that the loop body will run at least once, to compute *fib(2)*, as *i* is *1* and *n* is at least *2*; so the loop condition $i < n$ is *true*, which dictates that the loop body must be evaluated at least once.

Within the loop body we'll compute *fib(i+1)* (we call it *fib2*) by adding together *fib0* and *fib1*; then we increment *i*; then we update *fib0* and *fib1* so that for the *new* value of *i* they hold *fib(i-1)* and *fib(i)*. To do this we assign the initial value of *fib1* to *fib0* and the value of *fib2* to *fib1*. Think hard so as to confirm for yourself that this sequence of actions re-establishes the loop invariant.

Let's work an example. Suppose *n* happens to be *2*. The loop body will run, and after the one execution, *i* will have the value, *2*; *fib1* will have the value of *fib(2)*, and *fib0* will have the value of *fib(1)*. Because *i* is now *2* and *n* is *2*, the loop condition will now be false and the loop will terminate. The value of *fib1* will of course be *fib(i)* but now we also have the negation of the loop condition, i.e., $i == n$. So *fib(i)* will be *fib(n)*, which is the result we want and that we return.

We can also informally prove to ourself that this strategy gives us a program that always terminates and returns a value. That is, it does not go into an infinite loop. To see this, note that the value of *i* is initally less than or equal to *n*, and it increases by only *1* on each time through the loop. The value of *n* is finite, so the value of *i* will eventually equal the value of *n* at which point the loop condition will be falsified and the looping will end.

What Dafny looks for to verify that a given loop terminates is a value that *decreases* each time the loop runs and that is bounded below so that it cannot decrease forever. As *i* increases in this loop, it can not be the decreasing quantity. What Dafny takes instead is *n - i*. When *i* is *0* this value is large, and as *i* gets closer to *n* it decreases until when $i == n$, the difference is zero, and that is the bound at which the loop terminates.

That's our strategy. Here's the while loop that we have designed. Now for the first time, we see something crucial. In general, Dafny has no idea how many times a loop body will execute. Intead, what it needs to know are properties of the state of the program that hold no matter how many times the loop executes, that, when combined with the fact that the has terminated allows one to conclude that the loop does what it's meant to do. We call such properties *loop invariants*.

```
while (i < n)
    invariant i <= n;
    invariant fib0 == fib(i-1);
    invariant fib1 == fib(i);
{
    var fib2 := fib0 + fib1;
    fib0 := fib1;
    fib1 := fib2;
    i := i + 1;
}
```

The invariants are just the conditions that we required to hold for our design of the loop to work. First, *i* must never exceed *n*. If it did, the loop would spin off into infinity. Second, to compute the next (the *i+1st)* Fibonacci number we have to have the previous *two* in memory. So *fib0* better hold *fib(i-1)* and *fib1*, *fib(i)*. Note that these conditions do not have to hold at all times *within* the execution of the loop body, where things are being updated, but they do have to hold before before and after each execution.

The body of the loop is just as we described it above. We can use our minds to deduce that if the invariants hold before each loop body runs (and they do), then they will also hold after it runs. We can also see that after the loop terminates, it must be that *i==n*, because we know that it's always true that *i <= n* and the loop condition must now be false, which is to say that *i* can no longer be strictly less than *n*, so *i* must now equal *n*. Logic says so.

What is amazing is that we can write these assertions in Dafny if we wish to, and Dafny will verify that they are true statements about the state of the program after the loop has run. We have *proved* (or rather Dafny has proved) that our loop always terminates with the right answer. We have a formal proof of *total correctness* for this program.

```
assert i <= n;       // invariant
assert !(i < n);     // loop condition is false
assert (i <= n) && !(i < n) ==> (i == n);
assert i == n;       // deductive conclusion
assert fib1 == fib(i); // invariant
assert fib1 == fib(i) && (i==n) ==> fib1 == fib(n);
assert fib1 == fib(n);
return fib1;
```

5.8 What is Dafny, Again?

Dafny is a cutting-edge software language and toolset for verification of imperative code. It was developed at Microsoft Research—one of the top computer science research labs in the world. We are exploring Dafny and the ideas underlying it in the first part of this course to give a sense of why it's vital for a computer scientist today to have a substantial understanding of logic and proofs along with the ability to *code*.

Tools such as TLA+, Dafny, and others of this variety give us a way both to express formal specifications and imperative code in a unified way (albeit in different sub-languages), and to have some automated checking done in an attempt to verify that code satisfies its spec.

We say *attempt* here, because in general verifying the consistency of code and a specification is a literally unsolvable problem. In cases that arise in practice, much can often be done. It's not always easy, but if one requires ultra-high assurance of the consistency of code and specification, then there is no choice but to employ the kinds of *formal methods* introduced here.

To understand how to use such state-of-the-art software development tools and methods, one must understand not only the language of code, but also the languages of mathematical logic, including set and type theory. One must also understand precisely what it means to *prove* that a program satisfies its specification. And for that, one must develop a sense for propositions and proofs: what they are and how they are built and evaluated.

The well educated computer scientist and the professional software engineer must understand logic and proofs as well as coding, and how they work together to help build *trustworthy* systems. Herein lies the deep relevance of logic and proofs, which might otherwise seem like little more than abstract nonsense and a distraction from the task of learning how to program.

CHAPTER

SIX

6. DAFNY LANGUAGE: TYPES, STATEMENTS, EXPRESSIONS

6.1 Built-In Types

Dafny natively supports a range of abstract data types akin to those found in widely used, industrial imperative programming languages and systems, such as Python and Java. In this chapter, we introduce and briefly illustrate the use of these types. The types we discuss are as follow:

- bool, supporting Boolean algebra
- int, nat, and real types, supporting *exact* arithmetic (unlike the numerical types found in most industrial languages
- char, supporting character types
- set<T> and iset<T>, polymorphic set theory for finite and infinite sets
- seq<T> and iseq<T>, polymorphic finite and infinite sequences
- string, supporting character sequences (with addtional helpful functions)
- map<K,V> and imap<K,V>, polymorphic finite and infinite partial functions
- array<T>, polymorphic 1- and multi-dimensional arrays

6.1.1 Booleans

The bool abstract data type (ADT) in Dafny provides a bool data type with values, *true* and *false*, along with the Boolean operators that are supported by most programming langauges, along with a few that are not commonly supported.

Here's a method that computes nothing useful and returns no values, but that illustrates the range of Boolean operators in Dafny. We also use the examples in this chapter to discuss a few other aspects of the Dafny language.

```
method BoolOps(a: bool) returns (r: bool)  // bool -> bool
{
    var t: bool := true;    // explicit type declaration
    var f := false;         // type inferred automatically
    var not := !t;          // negation
    var conj := t && f;     // conjunction, short-circuit evaluation
    var disj := t || f;     // disjunction, short-circuit (sc) evaluation
    var impl := t ==> f;    // implication, right associative, sc from left
    var foll := t <== f;    // follows, left associative, sc from right
    var equv := t <==> t;   // iff, bi-implication
    return true;            // returning a Boolean value
 }
```

6.1.2 Numbers

Methods aren't required to return results. Such methods do their jobs by having side effects, e.g., doing output or writing data into global variables (usually a bad idea). Here's a method that doesn't return a value. It illustrates numerical types, syntax, and operations.

```
method NumOps()
{
    var r1: real := 1000000.0;
    var i1: int := 1000000;
    var i2: int := 1_000_000;   // underscores for readiability
    var i3 := 1_000;            // Dafny can often infer types
    var b1 := (10 < 20) && (20 <= 30); // a boolean expression
    var b2 := 10 < 20 <= 30;    // equivalent, with "chaining"
    var i4: int := (5.5).Floor; // 5
    var i5 := (-2.5).Floor;     // -3
    var i6 := -2.5.Floor;        // -2 = -(2.5.Floor); binding!
}
```

6.1.3 Characters

Characters (char) are handled sort of as they are in C, etc.

```
method CharFun()
{
    var c1: char := 'a';
    var c2 := 'b';
    // var i1 := c2 - c1;
    var i1 := (c2 as int) - (c1 as int);    // type conversion
    var b1 := c1 < c2;  // ordering operators defined for char
    var c3 := '\n';     // c-style escape for non-printing chars
    var c4 := '\u265B'; // unicode, hex, "chess king" character
}
```

6.1.4 Sets

Polymorphic finite and infinite set types: set<T> and iset<T>. T must support equality. Values of these types are immutable.

```
method SetPlay()
{
    var empty: set<int> := {};
    var primes := {2, 3, 5, 7, 11};
    var squares := {1, 4, 9, 16, 25};
    var b1 := empty < primes;    // strict subset
    var b2 := primes <= primes;   // subset
    var b3: bool := primes !! squares; // disjoint
    var union := primes + squares;
    var intersection := primes * squares;
    var difference := primes - {3, 5};
    var b4 := primes == squares;    // false
    var i1 := | primes |;   // cardinality (5)
    var b5 := 4 in primes;  // membership (false)
    var b6 := 4 !in primes; // non-membership
}
```

6.1.5 Sequences

Polymorphic sequences (often called "lists"): seq<T>. These can be understood as functions from indices to values. Some of the operations require that T support equality. Values of this type are immutable.

```
method SequencePlay()
{
    var empty_seq: seq<char> := [];
    var hi_seq: seq<char> := ['h', 'i'];
    var b1 := hi_seq == empty_seq; // equality; !=
    var hchar := hi_seq[0];        // indexing
    var b2 := ['h'] < hi_seq;   // proper prefix
    var b3 := hi_seq < hi_seq;  // this is false
    var b4 := hi_seq <= hi_seq; // prefix, true
    var sum := hi_seq + hi_seq; // concatenation
    var len := | hi_seq |;
    var hi_seq := hi_seq[0 := 'H']; // update
    var b5 := 'h' in hi_seq; // member, true, !in
    var s := [0,1,2,3,4,5];
    var s1 := s[0..2];  // subseqence
    var s2 := s[1..];   // "drop" prefix of len 1
    var s3 := s[..2];   // "take" prefix of len 2
    // there's a slice operator, too; later
 }
```

6.1.6 Strings

Dafny has strings. Strings are literally just sequences of characters (of type seq<char>), so you can use all the sequence operations on strings. Dafny provides additional helpful syntax for strings.

```
method StringPlay()
 {
     var s1: string := "Hello CS2102!";
     var s2 := "Hello CS2102!\n";   // return
     var s3 := "\"Hello CS2102!\""; // quotes
 }
```

6.1.7 Maps (Partial Functions)

Dafny also supports polymorphic maps, both finite (map<K,V>) and infinite (imap<K,V>). The key type, K, must support equality (==). In mathematical terms, a map really represents a binary relation, i.e., a set of <K,V> pairs, which is to say a subset of the product set, K * V, where we view the types K and V as defining sets of values.

```
method MapPlay()
{
    // A map literal is keyword map + a list of maplets.
    // A maplet is just a single <K,V> pair (or "tuple").
    // Here's an empty map from strings to ints
    var emptyMap: map<string,int> := map[];

    // Here's non empty map from strings to ints
    // A maplet is "k := v," k and v being of types K and V
    var aMap: map<string,int>  := map["Hi" := 1, "There" := 2];

    // Map domain (key) membership
```

```
    var isIn: bool := "There" in aMap; // true
    var isntIn := "Their" !in aMap;    // true

    // Finite map cardinality (number of maplets in a map)
    var card := |aMap|;

    //Map lookup
    var image1 := aMap["There"];
    // var image2 := aMap["Their"]; // error! some kind of magic
    var image2: int;
    if ("Their" in aMap) { image2 := aMap["Their"]; }

    // map update, maplet override and maplet addition
    aMap := aMap["There" := 3];
    aMap := aMap["Their" := 10];
}
```

6.1.8 Arrays

Dafny supports arrays. Here's we'll see simple 1-d arrays.

```
method ArrayPlay()
{
    var a := new int[10]; // in general: a: array<T> := new T[n];
    var a' := new int[10];   // type inference naturally works here
    var i1 := a.Length;      // Immutable "Length" member holds length of array
    a[3] := 3;           // array update
    var i2 := a[3];          // array access
    var seq1 := a[3..8];    // take first 8, drop first 3, return as sequence
    var b := 3 in seq1;     // true! (see sequence operations)
    var seq2 := a[..8];     // take first 8, return rest as sequence
    var seq3 := a[3..];     // drop first 3, return rest as sequence
    var seq4 := a[..];      // return entire array as a sequence
}
```

Arrays, objects (class instances), and traits (to be discussed) are of "reference" types, which is to say, values of these types are stored on the heap. Values of other types, including sets and sequences, are of "value types," which is to say values of these types are stored on the stack; and they're thus always treated as "local" variables. They are passed by value, not reference, when passed as arguments to functions and methods. Value types include the basic scalar types (bool, char, nat, int, real), built-in collection types (set, multiset, seq, string, map, imap), tuple, inductive, and co-inductive types (to be discussed). Reference type values are allocated dynamically on the heap, are passed by reference, and therefore can be "side effected" (mofified) by methods to which they are passed.

6.2 Statements

6.2.1 Block

In Dafny, you can make one bigger command from a sequence of smaller ones by enclosing the sequence in braces. You typically use this only for the bodies of loops and the parts of conditionals.

```
{
    print "Block: Command1\n";
```

```
    print "Block: Command2\n";
}
```

6.2.2 Break

The break command is for prematurely breaking out of loops.

```
var i := 5;
while (i > 0)
{
    if (i == 3)
    {
        break;
    }
    i := i - 1;
}
print "Break: Broke when i was ", i, "\n";
```

6.2.3 Update (Assignment)

There are several forms of the update command. The first is the usual assignment that you see in many languages. The second is "multiple assignment", where you can assign several values to several variables at once. The final version is not so familar. It *chooses* a value that satisfies some property and assigns it to a variable.

```
var x := 3;          // typical assignment
var y := 4;          // typical assignment
print "Update: before swap, x and y are ", x, ", ", y, "\n";
x, y := y, x;        // one-line swap using multiple assignment
print "Update: after swap, x and y are ", x, ", ", y, "\n";
var s: set<int> := { 1, 2, 3 }; // typical: assign set value to s
var c :| c in s;     // update c to a value such that c is in s
print "Update: Dafny chose this value from the set: ", c, "\n";
```

6.2.4 Var (variable declaration)

A variable declaration stsatement is used to declare one or more local variables in a method or function. The type of each local variable must be given unless the variable is given an initial value in which case the type will be inferred. If initial values are given, the number of values must match the number of variables declared. Note that the type of each variable must be given individually. This "var x, y : int;" does not declare both x and y to be of type int. Rather it will give an error explaining that the type of x is underspecified.

```
var l: seq<int> := [1, 2, 3]; // explicit type (sequence of its)
var l'          := [1, 2, 3]; // Dafny infers type from [1, 2, 3]
```

6.2.5 If (conditional)

There are several forms of the if statement in Dafny. The first is "if (Boolean) block-statement." The second is "if (Boolean) block-statement else block-statement" A block is a sequence of commands enclosed by braces (see above).

In addition, there is a multi-way if statement similar to a case statement in C or C++. The conditions for the cases are evaluated in an unspecified order. The first to match results in evaluation of the corresponding command. If no case matches the overall if command does nothing.

```
if (0==0) { print "If: zero is zero\n"; }   // if (bool) {block}
if (0==1)
    { print "If: oops!\n"; }
else
    { print "If: oh good, 0 != 1\n"; }

var q := 1;
if {
    case q == 0 => print "Case: q is 0\n";
    case q == 1 => print "Case: q is 1\n";
    case q == 2 => print "Case: q is 2\n";
}
```

6.2.6 While (iteration)

While statements come in two forms. The first is a typical Python-like statement "while (Boolean) block-command". The second involves the use of a case-like construct instead of a single Boolean expression to control the loop. This form is typically used when a loop has to either run up or down depending on the initial value of the index. An example of the first form is given above, for the BREAK statement. Here is an example of the second form.

```
var r: int;
while
    decreases if 0 <= r then r else -r;
{
    case r < 0 => { r := r + 1; }
    case 0 < r => { r := r - 1; }
}
```

Dafny insists on proving that all while loops and all recursive functions actually terminate – do not loop forever. Proving such properties is (infinitely) hard in general. Dafny often makes good guesses as to how to do it, in which case one need do nothing more. In many other cases, however, Dafny needs some help. For this, one writes "loop specifications." These include clauses called "decreases", "invariant", and "modifies", which are written after the while and before the left brace of the loop body. We discuss these separately, but in the meantime, here are a few examples.

```
// a loop that counts down from 5, terminating when i==0.
i := 5;                 // already declared as int above
while 0 < i
    invariant 0 <= i    // i always >= 0 before and after loop
    decreases i         // decreasing value of i bounds the loop
{
    i := i - 1;
}

// this loop counts *up* from i=0 ending with i==5
// notice that what decreases is difference between i and n
var n := 5;
i := 0;
while i < n
    invariant 0 <= i <= n
    decreases n - i
{
```

```
    i := i + 1;
}
```

6.2.7 Assert (assert a proposition about the state of the program)

Assert statements are used to express logical proposition that are expected to be true. Dafny will attempt to prove that the assertion is true and give an error if not. Once it has proved the assertion it can then use its truth to aid in following deductions. Thus if Dafny is having a difficult time verifying a method the user may help by inserting assertions that Dafny can prove, and whose true may aid in the larger verification effort. (From reference manual.)

```
assert i == 5;       // true because of preceding loop
assert !(i == 4);    // similarly true
// assert i == 4;    // uncomment to see static assertion failure
```

6.2.8 Print (produce output on console)

From reference manual: The print statement is used to print the values of a comma-separated list of expressions to the console. The generated C# code uses the System.Object.ToString() method to convert the values to printable strings. The expressions may of course include strings that are used for captions. There is no implicit new line added, so to get a new line you should include "n" as part of one of the expressions. Dafny automatically creates overrides for the ToString() method for Dafny data types.

```
print "Print: The set is ", { 1, 2, 3}, "\n"; // print the set
```

6.2.9 Return

From the reference manual: A return statement can only be used in a method. It terminates the execution of the method. To return a value from a method, the value is assigned to one of the named return values before a return statement. The return values act very much like local variables, and can be assigned to more than once. Return statements are used when one wants to return before reaching the end of the body block of the method. Return statements can be just the return keyword (where the current value of the out parameters are used), or they can take a list of values to return. If a list is given the number of values given must be the same as the number of named return values.

To return a value from a method, assign to the return parameter and then either use an explicit return statement or just let the method complete.

```
method ReturnExample() returns (retval: int)
{
    retval := 10;
    // implicit return here
}
```

Methods can return multiple values.

```
method ReturnExample2() returns (x: int, y:int)
{
    x := 10;
    y := 20;
}
```

The return keyword can be used to return immediatey

```
method ReturnExample3() returns (x: int)
{
    x := 5;      // don't "var" decare return variable
    return;      // return immediately
    x := 6;      // never gets executed
    assert 0 == 1; // can't be reached to never gets checked!
}
```

6.3 Expressions

6.3.1 Literals Expressions

A literal expression is a boolean literal (true or false), a null object reference (null), an unsigned integer (e.g., 3) or real (e.g., 3.0) literal, a character (e.g., 'a') or string literal (e.g., "abc"), or "this" which denote the current object in the context of an instance method or function. We have not yet seen objects or talked about instance methods or functions.

6.3.2 If (Conditional) Expressions

If expressions first evaluate a Boolean expression and then evaluate one of the two following expressions, the first if the Boolean expression was true, otherwise the second one. Notice in this example that an IF *expression* is used on the right side of an update/assignment statement. There is also an if *statement*.

```
var x := 11;
var h := if x != 0 then (10 / x) else 1;    // if expression
assert h == 0;
if (h == 0) {x := 3; } else { x := 0; }     // if statement
assert x == 3;
```

6.3.3 Conjunction and Disjunction Expressions

Conjunction and disjuction are associative. This means that no matter what b1, b2, and b3 are, (b1 && b2) && b3 is equal to (b1 && (b2 && b3)), The same property holds for ||.

These operators are also *short circuiting*. What this means is that their second argument is evaluated only if evaluating the first does not by itself determine the value of the expression.

Here's an example where short circuit evaluation matters. It is what prevents the evaluation of an undefined expressions after the && operator.

```
var a: array<int> := null;
var b1: bool := (a != null) && (a[0]==1);
```

Here short circuit evaluation protects against evaluation of a[0] when a is null. Rather than evaluating both expressions, reducing them both to Boolean values, and then applying a Boolean *and* function, instead the right hand expressions is evaluated "lazily", i.e., only of the one on the left doesn't by itself determine what the result should be. In this case, because the left hand expression is false, the whole expression must be false, so the right side not only doesn't have to be evaluated; it also *won't* be evaluated.

6.3.4 Sequence, Set, Multiset, and Map Expressions

Values of these types can be written using so-called *display* expressions. Sequences are written as lists of values within square brackets; sets, within braces; and multisets using "multiset" followed by a list of values within braces.

```
var aSeq: seq<int> := [1, 2, 3];
var aVal := aSeq[1];    // get the value at index 1
assert aVal == 2;       // don't forget about zero base indexing

var aSet: set<int> := { 1, 2, 3};   // sets are unordered
assert { 1, 2, 3 } == { 3, 1, 2};   // set equality ignores order
assert [ 1, 2, 3 ] != [ 3, 1, 2];   // sequence equality doesn't

var mSet := multiset{1, 2, 2, 3, 3, 3};
assert (3 in mSet) == true;         // in-membership is Boolean
assert mSet[3] == 3;                // [] counts occurrences
assert mSet[4] == 0;

var sqr := map [0 := 0, 1 := 1, 2 := 4, 3 := 9, 4 := 16];
assert |sqr| == 5;
assert sqr[2] == 4;
```

6.3.5 Relational Expressions

Relation expressions, such as less than, have a relational operator that compares two or more terms and returns a Boolean result. The ==, !=, <, >, <=, and >= operators are examples. These operators are also "chaining". That means one can write expressions such as 0 <= x < n, and what this means is 0 <= x && x < n.

The in and !in relational operators apply to collection types. They compute membership or non-membership respectively.

The !! operator computes disjointness of sets and multisets. Two such collections are said to be disjoint if they have no elements in common. Here are a few examples of relational expressions involving collections (all given within assert statements).

```
assert 3 in { 1, 2, 3 };                            // set member
assert 4 !in { 1, 2, 3 };                           // non-member
assert "foo" in ["foo", "bar", "bar"];              // seq member
assert "foo" in { "foo", "bar"};                    // set member
assert { "foo", "bar" } !! { "baz", "bif"};         // disjoint
assert { "foo", "bar" } < { "foo", "bar", "baz" };  // subset
assert { "foo", "bar" } == { "foo", "bar" };        // set equals
```

6.3.6 Array Allocation Expressions

Arrays in Dafny are *reference values*. That is, the value of an array variable is a *reference* to an address in the *heap* part of memory, or it is *null*. To get at the data in an array, one *dereferences* the array variable, using the *subscripting* operator. The array variable must not be null in this case. It must reference a chunk of memory that has been allocated for the array values, in the *heap* part of memory.

To allocate memory for a new array for n elements of type T one uses an expression like this: a: array<T> := new T[n]. The type of *a* here is "an array of elements of type *T*," and the size of the allocated memory chunk is big enough to hold *n* values of this type.

Multi-dimensional arrays (matrices) are also supported. The types of these arrays are "arrayn<T>, where "n" is the number of dimensions and T is the type of the elements. All elements of an array or matrix must be of the same type.

```
a := new int[10];       // type of a already declared above
var m: array2<int> := new int[10, 10];
a[0] := 1;              // indexing into 1-d array
m[0,0] := 1;            // indexing into multi-dimensional array
```

6.3.7 Old Expressions

An old expression is used in postconditions. old(e) evaluates to the value expression e had on entry to the current method. Here's an example showing the use of the old expression. This method increments (adds one **to_** the first element of an array. The specification part of the method *ensures* that the method body has this effect by explaining that the new value of a[0] must be the original (the "old") value plus one. The *requires* (preconditions) statements are needed to ensure that the array is not null and not zero length. The modifies command explains that the method body is allowed to change the value of a.

```
method incr(a: array<nat>) returns (r: array<nat>)
requires a != null;
requires a.Length > 0;
modifies a;
ensures a[0] == old(a[0]) + 1;
{
    a[0] := a[0] + 1;
    return a;
}
```

6.3.8 Cardinality Expressions

For a collection expression c, **|c|** is the cardinality of c. For a set or sequence the cardinality is the number of elements. For a multiset the cardinality is the sum of the multiplicities of the elements. For a map the cardinality is the cardinality of the domain of the map. Cardinality is not defined for infinite maps.

```
var c1 := | [1, 2, 3] |;            // cardinality of sequence
assert c1 == 3;
var c2 := | { 1, 2, 3 } |;          // cardinality of a set
assert c2 == 3;
var c3 := | map[ 0 := 0, 1 := 1, 2 := 4, 3 := 9] |; // of a map
assert c3 == 4;
assert | multiset{ 1, 2, 2, 3, 3, 3, 4, 4, 4, 4 } | == 10; // multiset
```

6.3.9 Let Expressions

A let expression allows binding of intermediate values to identifiers for use in an expression. The start of the let expression is signaled by the var keyword. They look like local variable declarations except the scope of the variable only extends to following expression. (Adapted from RefMan.)

Here's an example (see the following code).

First x+x is computed and bound to sum, the result of the overall expression on the right hand side of the update/assignment statement is then the value of "sum * sum" given this binding. The binding does not persist past the evaluation of the "let" expression. The expression is called a "let" expression because in many other languages, you'd use a let keyword to write this: let sum = x + x in sum * sum. Dafny just uses a slightly different syntax.

```
assert x == 3;                // from code above
var sumsquared := (var sum := x + x; sum * sum);  // let example
assert sumsquared == 36;      // because of the let expression
```

CHAPTER

SEVEN

7. SET THEORY

Modern mathematics is largely founded on set theory: in particular, on what is called *Zermelo-Fraenkel set theory with the axiom of Choice*, or *ZFC*. Every concept you have ever learned in mathematics can, in principle, be reduced to expresions involving sets. For example, every natural number can be represented as a set: zero as the *empty set, {}*; one as the set containing the empty set, *{{}}*; two as the set that contains that set, *{{{}}}*; ad infinitum.

Set theory includes the treatment of sets, including the special cases of relations (sets of tuples), functions (*single-valued* relations), sequences (functions from natural numbers to elements), and other such concepts. ZFC is a widely accepted *formal foundation* for modern mathematics: a set of axioms that describe properties of sets, from which all the rest of mathematics can be deduced.

7.1 Naive Set Theory

So what is a set? A *naive* definition (which will actually be good enough for our purposes and for most of practical computer science) is that a set is just an unordered collection of elements. In principle, these elements are themselves reducible to sets but we don't need to think in such reductionist terms. We can think about a set of natural numbers, for example, without having to think of each number as itself being some weird kind of set.

In practice, we just think sets as unordered collections of elements of some kind, where any given element is either *in* or *not in* any given set. An object can be a member of many different sets, but can only by in any give set zero or one times. Membership is binary. So, for example, when we combine (take the *union* of) two sets, each of which contains some common element, the resulting combined set will have that element as a member, but it won't have it twice.

This chapter introduces *naive*, which is to say *intuitive and practical*, set theory. It does not cover *axiomatic* set theory, in which every concept is ultimately reduced to a set of logical axioms that define what precisely it means to be a set and what operations can be use to manipulate sets.

7.2 Overly Naive Set Theory

Before we go on, however, we review a bit of history to understand that an overly naive view of sets can lead to logical contradictions that make such a theory useless as a foundation for mathematics.

One of the founders of modern logic, Gotlob Frege, had as his central aim to establish logical foundations for all of mathematics: to show that everything could be reduced to a set of axioms, or propositions accepted without question, from which all other mathematical truths could be deduced. The concept of a set was central to his effort. His logic therefore allowed one to define sets as collections of elements that satisfy given propositions, and to talk about whether any given element is in a particular set of not. Frege's notion of sets, in turn, traced back to the work of Georg Cantor.

But then, boom! In 1903, the British analytical philosopher, Bertrand Russell, published a paper presenting a terrible paradox in Frege's conception. Russell showed that a logic involving naive set theory would be *inconsistent* (self-contradicting) and there useless as a foundation for mathematics.

To see the problem, one consider the set, *S*, of all sets that do not contain themselves. In *set comprehension* notation, we would write this set as $S = \{a : set | a \notin a\}$. That is, *S* is the set of elements, *a*, each a set, such that *a* is not a member of itself.

Now ask the decisive question: Does *S* contain itself?

Let's adopt a notation, *C(S)*, to represent the proposition that *S* contains itself. Now suppose that *C(S)* is true, i.e., that *S* does contain itself. In this case, *S*, being a set that contains itself, cannot be a member of *S*, because we just defined *S* to be the set of sets that do *not* contain themselves. So, the assumption that *S* contains itself leads to the conclusion that *S* does not contain itself. In logical terms, $C(S) \rightarrow \neg C(S)$. This is a contradiction and thus a logical impossibility.

Now suppose *S* does not contain itself: $\neg C(S)$.. Being such a set, and given that *S* is the set of sets that do not contain themselves, it must now be in *S*. So $\neg C(S) \rightarrow C(S)$. The assumption that it does *not* contain itself leads right back to the conclusion that it *does* contain itself. Either the set does or does not contain itself, but assuming either case leads to a contradictory conclusion. All is lost!

That such an internal self-contradiction can arise in such a simple way (or at all) is a complete disaster for any logic. The whole point of a logic is that it gives one a way to reason that is sound, which means that from true premises one can never reach a contradictory conclusion. If something that is impossible can be proved to be true in a given theory, then anything at all can be proved to be true, and the whole notion of truth just collapses into meaninglessness. As soon as Frege saw Russell's Paradox, he knew that that was *game over* for his profound attempt to base mathematics on a logic grounded in his (Cantor's) naive notion of sets.

Two solutions were eventually devised. Russell introduced a notion of *types*, as opposed to sets, per se, as a foundation for mathematics. The basic idea is that one can have elements of a certain *type*; then sets of elements of that type, forming a new type; then sets of sets elements of that type, forming yet another type; but one cannot even talk about a set containing (or not containing) itself, because sets can only contain elements of types lower in the type hierarchy.

The concept of types developed by Russell lead indirectly to modern type theory, which remains an area of very active exploration in both computer science and pure mathematics. Type theory is being explored as an alternative foundation for mathematics, and is at the very heart of a great deal of work going on in the areas of programming language design and formal software specification and verification.

On the other hand, Zermelo repaired the paradox by adjusting some of the axioms of set theory, to arrive at the starting point of what has become ZFC. When we work in set theory today, whether with a *naive* perspective or not, we are usually working in a set theory the logical basic of which is ZFC.

7.3 Sets

For our purposes, the *naive* notion of sets will be good enough. We will take a *set* to be an unordered finite or infinite collection of *elements*. An element is either *in* or *not in* a set, and can be in a set at most once. In this chapter, we will not encounter any of the bizarre issues that Russell and others had to consider at the start of the 20th century.

What we will find is that set-theoretical thinking is an incredibly powerful intellectual tool. It's at the heart of program specification and verification, algorithm design and analysis, and theory of computing, among many other areas in computer science. Moreover, Dafny makes set theory not only fun but executable. The logic of Dafny, for writing assertions, pre- and post-conditions, and invariants *is* set theory, a first-order logic with sets and set-related operations as built-in concepts.

7.4 Set Theory Notations

7.4.1 Display notation

In everyday mathematical writing, and in Dafny, we denote small sets by listing the elements of the set within curly brace. If *S* is the set containing the numbers, one, two, and three, for example, we can write *S* as $\{1, 2, 3\}$.

In Dafny, we would write almost the same thing.

```
var S:set<int> := { 1, 2, 3 };
```

This code introduces the variable, *S*, declares that its type is *finite set of integer* (*iset<T>* being the type of *infinite* sets of elements of tyep *T*), and assigns to *S* the set value, $\{1, 2, 3\}$. Because the value on the right side of the assignment operator, is evidently a set of integers, Dafny will infer the type of *S*, and the explicit type declaration can therefore be omitted.

```
var S := { 1, 2, 3 };
```

When a set is finite but too large to write down easily as a list of elements, but when it has a regular structure, mathematicians often denote such a set using an elipsis. For example, a set, *S*, of even natural numbers from zero to one hundred could be written like this: $S = \{0, 2, 4, \ldots, 100\}$. This expression is a kind of quasi-formal mathematics. It's mostly formal but leaves details that an educated person should be able to infer to the human reader.

It is not (currently) possible to write such expressions in Dafny. Dafny does not try to fill in missing details in specifications. A system that does do such a thing might make a good research project. On the other hand, ordinary mathematical writing as well as Dafny do have ways to precisely specify sets, including even infinite sets, in very concise ways, using what is called *set comprehension* or *set builder* notation.

7.4.2 Set comprehension notation

Take the example of the set, *S*, of even numbers from zero to one hundred, inclusive. We can denote this set precisely in mathematical writing as $S = \{n : \mathbb{Z} \mid 0 <= n <= 100 \wedge n \; mod \; 2 = 0\}$. Let's pull this expression apart.

The set expression (to the right of the first equals sign) can be read in three parts. The vertical bar is read *such that*. To the left of the bar is an expression identifying the set from which the elements of *this* set are drawn, and a name is given to an arbitrary element of this source set. So here we can say that *S* is a set each element *n* of which is a natural number. A name, here *n*, for an arbitrary element is given for two purposes. First it desribes the form of elements in the set being built: here just *integers*. Second, the name can then be used in writing a condition that must be true of each such element. That expression is written to the right of the vertical bar.

Here the condition is that each such element, *n* must be greater than or equal to zero, less than or equal to one hundred, and even, in that the remainder must be zero when *n* is divided by *2*. The overall set comprehension expression is thus read literally as, *S* is the set of integers, *n*, such that *n* is greater than or equal to zero, *n* is less than or equal to 100, and *n* evenly divisible by *2*. A more fluent reading would simply be *S* is the set of even integers between zero and one hundred inclusive.

Dafny supports set comprehension notations. This same set would be written as follows (we assume that the type of S has already been declared to be *set<int>)*:

```
S := set s: int | 0 <= s <= 100;
```

Another way to define the same set in ordinary mathematical writing would use a slightly richer form of set comprehension notation. In particular, we can define the same set as the set of values of the expression *2*n* for *n* is in the range zero to fifty, inclusive. Where it's readily inferred, mathematicians will usually also leave out explicit type information. ‘S = { 2 * n | 0 <= n <= 50 }. In this expression it's inferred that *n* ranges over all the natural numbers,

these values are *filtered* by the expression on the right, and these filtered values are then fed through the expression on the left of the bar to produce the elements of the intended set.

Dafny also supports set comprehension notation in this style. To define this very same set in Dafny we could also write this:

```
S := set s: int | 0 <= s <= 50 :: 2 * s;
```

This command assigns to S a set of values, *2 * s*,, where *s* ranges over the integers and satisfies the predicate (or filter) *0 <= s <= 50*.

The collection of values from which element are drawn to be build into a new set need not just be a built-in type but can be another programmer-defined set. Given that *S* is the set of even numbers from zero to one hundred, we can define the subset of *S* of elements that are less than *25* by writing a richer set comprehension. In pure mathematical writing, we could write $T = \{t | t \in S \wedge t < 25\}$. That is, *T* is the set of elements that are in *S* and less than *25*. The Dafny notation is a little different, but not too much:

```
var T := set t | t in S && t < 25;
```

This Dafny code defines *T* to be the set (of integers, but note that we let Dafny infer the type of *t* in this case), such that *t* is in the set *S* (that we just defined) and *t* is also less than *25*.

As a final example, let's suppose that we want to define the set of all ordered pairs whose first elements are from *S* and whose second elements are from *T*, as we've defined them here. For example, the pair *(76,24)* would be in this set, but not *(24 76)*. In ordinary mathematical writing, this would be $\{(s, t) | s \in S \wedge t \in T\}$. This set is, as we'll learn more about shortly, called the *product set* of the sets, *S* and *T*.

In Danfy, this would be written like this:

```
var Q := set s, t | s in S && t in T :: (s, t);
```

This code assigns to the new variable, *Q*, a set formed by taking elements, *s* and *t*,, such that *s* is in *S* and *t* is in *T*, and forming the elements of the new set as tuples, *(s, t)*. This is a far easier way to write code for a product set than by explicit iteration over the sets *S* and *T*!

In Dafny, the way to extract an element of a tuple, *t*, of arity, *n*, is by writing *t.n*, where *n* is a natural number in the range *0* up to *n - 1*. So, for example, *(3, 4).1* evaluates to *4*. It's not a notation that is common to many programming languages. One can think of it as a kind of subscripting, but using a different notation than the usual square bracket subscripting used with sequences.

7.5 Set Operations

7.5.1 Cardinality

By the cardinality of a set, *S*, we mean the number of elements in S. When *S* is finite, the cardinality of *S* is a natural number. The cardinarily of the empty set is zero, for example, because it has no (zero) elements. In ordinary mathematics, if *S* is a finite set, then its cardinality is denoted $|S|$. With *S* defined as in the preceding section, the cardinality of *S* is *50*. (There are *50* numbers between *0* and *49*, inclusive.)

The Dafny notation for set cardinality is just the same. The following code will print the cardinality of *S*, namely *50*, for example.

```
print |S|;
```

If a set is infinite in size, as for example is the set of natural numbers, the cardinality of the set is obviously not any natural number. One has entered the realm of *transfinite numbers*. We will discuss transfinite numbers later in this course. In Dafny, as you might expect, the cardinality operator is not defined for infinite sets (of type *iset<T>*).

7.5.2 Equality

Two sets are considered equal if and only if they contain exactly the same elements. To assert that sets *S* and *T* are equal in mathematical writing, we would write *S* = *T*. In Dafny, such an assertion would be written, *S* == *T*.

7.5.3 Subset

A set, *T*, can be said to be a subset of a set *S* if and only if every element in *T* is also in *S*. In this case, mathematicians write $T \subseteq S$. In mathematical logic notation, we would write, $T \subseteq S \iff \forall t \in T, t \in S$. That is, *T* is a subset of *S* if and only if every element in *T* is also in *S*.

A set *T*, is said to be a *proper* subset of *S*, if *T* is a subset of *S* but *T* is not equal to *S*. In our example, *T* (the set of even natural numbers less than *25*) is a proper subset of *S* (the set of even natural numbers less than or equal to *100*).

This is written in mathematics as $T \subset S$. In other words, every element of *T* is in *S* but there is at least one element of *S* that is not in *T*. Mathematically, $T \subset S \iff \forall t \in T, t \in S \wedge \exists s \in S, s \notin T$.

The backwards *E* is the *existential quantifier* in first-order logic, and is read as, and means, *there exists.* So this expression says that *T* is a proper subset of *S* if every *t* in *T* is in *S* but there is at least one *s* in *S* that is not in *T*. That the proper subset operator contains an implicit existential operator poses some real problems for verification.

Without getting into details, when one asserts in Dafny that *T* is a proper subset of *S*, Dafny needs to find an element of *S* that is not in *T*, and in general, it needs a lot of help to do that. The details are out of scope at this point, but one should be aware of the difficulty.

In Dafny, one uses the usual arithmetic less and less than or equal operator symbols, < and <=, to assert *proper subset* and *subset* relationships, respectively. The first two of the following assertions are thus both true in Dafny, but the third is not. That said, limitations in the Dafny verifier make it hard for Dafny to see the truth of such assertions without help. We will not discuss how to provide such help at this point.

```
assert T < S;
assert T <= S;
assert S <= T;
```

We note every set is a subset, but not a proper subset, of itself. It's also the case that the empty set is a subset of every set, in that *all* elements in the empty set are in any other set, because there are none. In logic-speak, we'd say *a universally quantified proposition over an empty set is trivially true.*

If we reverse the operator, we get the notion of supersets and proper supersets. If *T* is a subset of *S*, then *S* is a superset of *T*, written, $S \supseteq T$. If *T* is a proper subset of *S* then *S* is a proper superset of *T*, written $S \supset T$. In Dafny, the greater than and greater than or equals operator are used to denote proper superset and superset relationships between sets. So, for example, *S* >= *T* is the assertion that *S* is a superset of *T*. Note that every set is a superset of itself, but never a proper superset of itself, and every set is a superset of the empty set.

7.5.4 Intersection

The intersection, $S \cap T$, of two sets, *S* and *T*, is the set of elements that are in both *S* and *T*. Mathematically speaking, $S \cap T = \{e \mid e \in S \wedge e \in T\}$.

In Dafny, the * operator is used for set intersection. The intersection of *S* and *T* is thus written *S* * *T*. For example, the command *Q* := *S* * *T* assigns the intersection of *S* and *T* as the new value of *Q*.

7.5.5 Union

The union, $S \cup T$, of two sets, *S* and *T*, is the set of elements that are in either (including both) *S* and *T*. That is, $S \cup T = \{e \mid e \in S \vee e \in T\}$.

In Dafny, the + operator is used for set union. The union of *S* and *T* is thus written *S + T*. For example, the command *V := S + T* assigns the union of *S* and *T* as the new value of *V*.

7.5.6 Difference

The difference, $S \setminus T$ (*S* minus *T*), of sets *S* and *T* is the set of elements in *S* that are not also in *T*. Thus, $S \setminus T = \{e \mid e \in S \wedge e \notin T)$. In Dafny, the minus sign is used to denote set difference, as in the expression, *S - T*. Operators in Dafny can be applied to sets to make up more complex expressions. So, for example, |*S-T*| denotes the cardinality of *S-T*.

7.5.7 Product Set

The product set, $S \times T$, is the set of all the ordered pairs, *(s,t)*, that can be formed by taking one element, *s*, from *S*, and one element, *t*, from *T*. That is, $S \times T = \{(s,t)|s \in S \wedge t \in T\}$. The cardinality of a product set is the product of the cardinalities of the individual sets.

There is no product set operator, per se, in Dafny, but given sets, *S* and *T* a product set can easily be expressed using Dafny's set comprehension notation: *set s, t | s in S && t in T :: (s,t)*. The keyword, *set*, is followed by the names of the variables that will be used to form the set comprehension expression, followed by a colon, followed by an assertion that selects the values of *s* and *t* that will be included in the result, followed by a double colon, and then, finally an expression using the local variables that states how each value of the resulting set will be formed.

7.5.8 Power Set

The power set of a set, *S*, denoted $\mathbb{P}(S)$, is the set of all subsets of *S*. If *S = {1, 2 }*, for example, the powerset of *S* is the set containing the proper and improper subsets of *S*, namely *{}, { 1 }, { 2 },* and *{ 1, 2}*.

The powerset of a set with *n* element will have 2^n elements. Consider the powerset of the empty set. The only subset of the empty set is the empty set itself, so the powerset of the empty set is the set containing only the empty set. This set has just *1* element. It's cardinality thus satisfies the rule, as *2* to the power, zero (the number of elements in the empty set), is *1*.

Now suppose that for every set, *S*, with cardinality *n*, the cardinality of its powerset is *2* to the *n*. Consider a set, *S'*, of cardinality one bigger than that of *S*. Its powerset contains every set in the powerset of *S*, plus every set in that set with the new element included, and that's all the element it includes.

The number of sets in the powerset of *S'* is thus double the number of sets in the powerset of *S*. Given that the cardinality of the powerset of *S* is *2* to the *n*, the cardinality of *S'*, being twice that number, is *2* to the *n + 1*.

Now because the rule holds for sets of size zero, and whenver it holds for sets of size *n* it also holds for sets of size *n + 1*, it must hold for sets of every (finite) size. So what we have is an informal *proof by induction* of a theorem: $\forall S, |\mathbb{P}(S)| = 2^{|S|}$.

In Dafny, there is no explicit powerset operator, one that would take a set and returning its powerset, but the concept can be expressed in an elegant form using a set comprehension. The solution is simply to say *the set of all sets that are subsets of a given set, **. In pure mathematical notation this would be $R|R \subseteq S$. In Dafny it's basically the same expression. The follwing three-line program computes and prints out the powerset of *S = { 1, 2, 3 }*. The key expression is to the right of the assignment operator on the second line.

```
var S := { 1, 2, 3 };
var P := set R | R <= S;
print P;
```

Exercise: Write a pure function that when given a value of type set<T> returns its powerset. The function will have to be polymorphic. Call it powerset<T>.

7.6 Tuples

A tuple is an ordered collection of elements. The type of elements in a tuple need not all be be the same. The number of elements in a tuple is called its *arity*. Ordered pairs are tuples of arity, *2*, for example. A tuple of arity *3* can be called a (an ordered) *triple*. A tuple of a larger arity, *n*, is called an *n-tuple*. The tuple, *(7, X, "house", square_func)*, for example, is a *4-tuple*.

As is evident in this example, the elements of a tuple are in general not of the same type, or drawn from the same sets. Here, the first element is an integer; the second, a variable;, the third, a string; and last, a function.

An *n*-tuples should be understood as values taken from a product of *n* sets. If *S* and *T* are our sets of even numbers between zero and one hundred, and zero and twenty four, for example, then the ordered pair, *(60,24)* is an element of the product set $S \times T$. The preceding *4*-tuple would have come from a product of four sets: one of integers, one of variables, one of strings, and one of functions.

The *type* of a tuple is the tuple of the types of its elements. In mathematical writing, we'd say that the tuple, *(-3,4)* is al element of the set $\mathbb{Z} \times \mathbb{Z}$, and if asked about its type, most mathematicians would say *pair of integers*. In Dafny, where types are more explicit than they usually are in quasi-formal mathematical discourse, the type of this tuple is *(int, int)*. In general, in both math and in Dafny, in particular, the type of a tuple in a set product, ::*S_1 times S_2 times ldots time S_n*, where the types of these sets are $T_1, \ldots, T_n$ is $(T_1, \ldots, T_n)$.

The elements of a tuple are sometimes called *fields of that tuple. Given an *n*-tuple, *t*, we are often interested in working with the value of one of its fields. We thus need a function for *projecting* the value of a field out of a tuple. We actually think of an *n*-tuple as coming with *n* projection functions, one for each field.

Projection functions are usually written using the Greek letter, ::*pi*, with a natural number subscript indicating which field a given projection function " projects". Given a *4*-tuple, *t = (7, X, "house", square_func)*, we would have math::*pi_0(t) = 7* and $\pi_3(t) = square_func$.

The type of a projection funcion is *function from tuple type to field type*. In general, because tuples have fields of different types, they will also have projection functions of different types. For example, pi_0 here is of type (in Dafny) $(int, variable, string, int \to int) \to \mathbb{Z}$ while pi_3 is of type $(int, variable, string, int \to int) \to (int \to int)$.

In Dafny, tuples are written as they are in mathematics, as lists of field values separated by commas and enclosed in parentheses. For example *t := (1, "hello", [1,2,3])" assigns to *t* a *3-tuple* whose first field has the value, *1* (of type *int*); whose second field has the value, "hello", a string; and whose third element is the list of integers, *[2, 4, 6]*.

Projection in Dafny is accomplished using the *tuple* subscripting (as opposed to array or list subscripting) operation. Tuple subscripting is done by putting a dot (period) followed by an index after the tuple expression. Here's a little Dafny code to illustrate. It defines *t* to be the triple, *(7, 'X', "hello")* (of type *(int, char, string)*), and then usses the *.0* and *.2* projection functions to project the first and third elements of the tuple, which it prints. To make the type of the tuple explicit, the final line of code declare *t'* to be the same tuple value, but this time explicitly declares its type.

```
var t := (7, 'X', "hello");
print t.0;
print t.2;
var t': (int, char, string) := (7, 'X', "hello");
```

While all of this might seem a little abstract, it's actually simple and very useful. Any table of data, such as a table with columns that hold names, birthdays, and social security numbers, represents data in a product set. Each row is a

tuple. The columns correspond to the sets from which the field values are drawn. One set is a set of names; the second, a set birthdays; the third, a set of social security numbers. Each row is just a particular tuple in product of these three sets, and the table as a whole is what we call a *relation*. If you have heard of a *relational database*, you now know what kind of data such a system handles: tables, i.e., *relations*.

7.7 Relations

A relation in nothing but a subset of (the tuples in) a product set. A table such as the one just described, will, in practice, usually not have a row with every possible combination of names, birthdays, and SSNs. In other words, it won't be the entire product of the sets from which the field values drawn. Rather, it will usually contain a small subset of the product set.

In mathematical writing, we will thus often see a sentence of the form, Let $R \subseteq S \times T$ be a (binary) relation on *S* and *T*. All this says is that *R* is some subset of the set of all tuples in the product set of *S* and *T*. If *S = { hot, cold }* and *T = { cat, dog }*, then the product set is *{ (hot, cat), (hot, dog), (cold, cat), (cold, dog) }*, and a relation on *S* and *T* is any subset of this product set. The set, *{ (hot, cat), (cold, dog) }* is thus one such relation on *S* and *T*.

Here's an exercise. If *S* and *T* are finite sets, with cardinalities |*S*| = *n* and |*T*| = *m*, how many relations are there over *S* and *T*? Hint: First, how many tuples are in the product set? Second, how many subsets are there of that set? For fun, write a little Dafny program that takes two sets of integers as arguments as return the number of relations over them. Write another function that takes two sets and returns the set of all possible relations over the sets. Use a set comprehension expression rather than writing a while loop. Be careful: the number of possible relations will be very large even in cases where the given sets contain only a few elements each.

7.8 Binary Relations

Binary relations, which play an especially important role in mathematics and computer science, are relations over just *2* sets. Suppose $R \subseteq S \times T$ is a binary relation on *S* and *T*. Then *S* is called the *domain* of the relation, and *T* is called its *co-domain*. That is, a binary relation is a subset of the ordered pairs in a product of the given domain and codomain sets.

If a particular tuple, *(s, t)* is an element of such a relation, *R*, we will say *R* is *defined for* the value, s, and that *R achieves* the value, *t*. The *support* of a relation is the subset of values in the domain on which it is defined. The *range* of a relation is the subset of co-domain values that it achieves.

For example, if *S = { hot, cold }* and *T = { cat, dog }*, and *R = *{ (hot, cat), (hot, dog) }*, then the domain of *R* is *S*; the co-domain of *R* is *T*; the support of *R* is just *{ hot }* (and *R* is thus *not defined* for the value *cold*); and the range of *R* is the whole co-domain, *T*.

The everyday functions you have studies in mathematics are binary relations, albeit usually infinite ones. For example, the *square* function, that associates every real number with its square, can be understood as the infinite set of ordered pairs of real numbers in which the second is the square of the first. Mathematically this is :{ (x, y) | y = x^2 }:‘, where we take as implicit that *x* and *y* range over the real numbers. Elements of this set include the pairs, *(-2, 4)* and *(2, 4)*.

The concept of *square roots* of real numbers is also best understood as a relation. The tuples are again pairs of real numbers, but now the elements include tuples, *(4, 2)* and *(4, -2)*.

7.8.1 Inverse

The inverse of a given binary relation is simply the set of tuples formed by reversing the order of all of the given tuples. To put this in mathematical notation, if *R* is a relation, its inverse, denoted R^{-1}, is $\{(y, x)|(x, y) \in R\}$. You can see this immediately in our example of squares and square roots. Each of these relations is the inverse of the other. One contains the tuples, *(-2, 4), (2, 4)*, while the other contains *(4, 2), (4, -2)*.

It should immediately be clear that the inverse of a function is not always also a function. The inverse of the *square* function is the *square root* relation, but that relation is not itself a function, because it is not single valued.

Here's a visual way to think about these concept. Consider the graph of the *square* function. Its a parabola that opens either upward in the *y* direction, or downward. Now select any value for *x* and draw a vertical line. It will intersect the parabola at only one point. The function is single-valued.

The graph of a square root function, on the other hand, is a parabola that opens to the left or right. So if one draws a vertial line at some value of *x*, either the line fails to hit the graph at all (the square root function is not defined for all values of *x*), or it intersects the line at two points. The square root "function" is not single-valued, and isn't really even a *function* at all. (If the vertical line hits the parabola right at its bottom, the set of points at which it intersects contains just one element, but if one takes the solution set to be a *multi-set*, then the value, zero, occurs in that set twice.)

A function whose inverse is a function is said to be *invertible*. The function, *f(x)* = *x* (or *y* = *x* if you prefer) is invertible in this sense. In fact, its inverse is itself.

Exercise: Is the cube root function invertible? Prove it informally.

Exercise: Write a definition in mathematical logic of what precisely it means for a function to be invertible. Model your definition on our definition of what it means for a relation to be single valued.

7.9 Functions: *Single-Valued* Relations

A binary-relation is said to be *single-valued* if it does not have tuples with the same first element and different second elements. A single-valued binary relation is also called a *function*. Another way to say that *R* is single valued is to say that if *(x, y)* and *(x, z)* are both in *R* then it must be that *y* and *z* are the same value. Otherwise the relation would not be single-valued! To be more precise, then, if $R \subseteq S \times T$, is single valued relation, then $(x, y) \in R \wedge (x, z) \in R \rightarrow y = z$.

As an example of a single-valued relation, i.e., a function, consider the *square*. For any given natural number (in the domain) under this function there is just a *single* associated value in the range (the square of the first number). The relation is single-valued in exactly this sense. By contrast, the square root relation is not a function, because it is not single-valued. For any given non-negative number in its domain, there are *two* associated square roots in its range. The relation is not single-valued and so it is not a function.

There are several ways to represent functions in Dafny, or any other programming language. One can represent a given function *implicity*: as a *program* that computes that function. But one can also represent a function *explicitly*, as a relation: that is, as a set of pairs. The (polymorphic) *map* type in Dafny provides such a representation.

A "map", i.e., a value of type *map<S,T>* (where *S* and *T* are type parameters), is to be understood as an explicit representation of a single-valued relation: a set of pairs: a function. In addition to a mere set of pairs, this data type also provides helpful functions and a clever representation underlying representation that both enforce the single-valuedness of maps, and that make it very efficient to look up range values given domain values where the map is defined, i.e., to *apply* such a function to a domain value (a "key") to obtained the related range *value*.

Given a Dafny map object, *m*, of type *map<S,T>*, one can obtain the set of values of type *S* for which the map is defined as *m.Keys()*. One can obtain the range, i.e., the set of values of type *T* that the map maps *to*, as *m.Values()*. One can determine whether a given key, *s* of type *S* is defined in a map with the expression, *s in m*.

Exercise: Write a method (or a function) that when given a map<S,T> as an argument returns a set<(T,S)> as a result where the return result represents the *inverse* of the map. The inverse of a function is not necessarily a function so the inverse of a map cannot be represented as a map, in general. Rather, we represent the inverse just as a *set* of *(S,T)* tuples.

Exercise: Write a pure function that when given a set of ordered pairs returns true if, viewed as a relation, the set is also a function, and that returns false, otherwise.

Exercise: Write a function or method that takes a set of ordered pairs with a pre-condition requiring that the set satisfy the predicate from the preceding exercise and that then returns a *map* that contains the same set of pairs as the given

set.

Exercise: Write a function that takes a map as an argument and that returns true if the function that it represents is invertible and that otherwise returns false. Then write a function that takes a map satisfying the precondition that it be invertible and that in this case returns its inverse, also as a map.

7.10 Properties of Functions

We now introduce essential concepts and terminology regarding for distinguishing essential properties and special cases of functions.

7.10.1 Total vs Partial

A function is said to be *total* if every element of its domain appears as the first element in at least one tuple, i.e., its *support* is its entire *domain*. A function that is not total is said to be *partial*. For example, the square function on the real numbers is total, in that it is defined on its entire real number domain. By contrast, the square root function is not total (if it domain is taken to be the real numbers) because it is not defined for real numbers that are less than zero.

Note that if one considers a slightly different function, the square root function on the *non-negative* real numbers the only difference being in the domainm then this function *is* total. Totality is thus relative to the specified domain. Here we have two functions with the very same set of ordered pairs, but one is total and the other is not.

Exercises: Is the function $y = x$ on the real numbers total? Is the *log* function defined on the non-negative real numbers total? Answer: no, because it't not defined at $x = 0$. Is the *SSN* function, that assigns a U.S. Social Security Number to every person, total? No, not every person has a U.S. Social Security number.

Implementing partial functions as methods or pure function in software presents certain problems. Either a pre-conditions has to be enforced to prevent the function or method being called with a value for which it's not defined, or the function or method needs to be made total by returning some kind of *error* value if it's called with such a value. In this case, callers of such a function are obligated always to check whether *some* validfunction value was returned or whether instead a value was returned that indicates that there is *no such value*. Such a value indicates an *error* in the use of the function, but one that the program caught. The failure of programmers systematically to check for *error returns* is a common source of bugs in real software.

Finally we note that by enforcing a requirement that every loop and recursion terminates, Dafny demands that every function and method be total in the sense that it returns and that it returns some value, even it it's a value that could flag an error.

When a Dafny total function is used to implement a mathematical function that is itself partial (e.g., *log(x)* for any real number, x), the problem thus arises what to return for inputs for which the underlying mathematical function is not defined. A little later in the course we will see a nice way to handle this issue using what are called *option* types. An option type is like a box that contains either a good value or an error flag; and to get a good value out of such a box, one must explicitly check to see whether the box has a good value in it or, alternatively, and error flag.

7.10.2 Injective

A function is said to be *injective* if no two elements of the domain are associated with the same element in the co-domain. (Note that we are limiting the concept of injectivity to functions.) An injective function is also said to be *one-one-one*, rather than *many-to-one*.

Take a moment to think about the difference between being injective and single valued. Single-valued means no *one* element of the domain "goes to" *more than one" value in the range. Injective means that "no more than one" value in the domain "goes to" and one value in the range.

Exercise: Draw a picture. Draw the domain and range sets as clouds with points inside, representing objects (values) in the domain and co-domain. Represent a relation as a set of *arrows* that connect domain objects to co-domain objects. The arrows visually depict the ordered pairs in the relation. What does it look like visually for a relation to be single-valued? What does it look like for a relation to be injective?

The square function is a function because it is single-valued, but it is not injective. To see this, observe that two different values in the domain, *-2* and *2*, have the same value in the co-domain: *4*. Think about the graph: if you can draw a *horizontal* line for any value of *y* that intersects the graph at multiple points, then the points at which it intersects correspond to different values of *x* that have the same value *under the relation*. Such a relation is not injective.

Exercises: Write a precise mathematical definition of what it means for a binary relation to be injective. Is the cube root function injective? Is *f(x) = sin(x)* injective?

An Aside: Injectivity in Type Theory

As an aside, we note that the concept of injectivity is essential in *type theory*. Whereas *set theory* provides a universally accepted axiomatic foundation for mathematics, *type* theory is of increasing interest as alternative foundation. It is also at the very heart of a great deal of work in programming languages and software verification.

Type theory takes types rather than sets to be elementary. A type in type theory comprises a collection of objects, just as a set does in set theory. But whereas in set theory, an object can be in many sets, in type theory, and object can have only one type.

The set of values of a given type is defined by a set of constants and functions called constructors. Constant constructors define what one can think of as the *smallest* values of a type, while constructors that are functions provide means to build larger values of a type by *packaging up" smaller values of the same and/or other types.

As a simple example, one might say that the set of values of the type, *Russian Doll,* is given by one constant constructor, *SolidDoll* and by one constructor function, *NestDoll* that takes a nested doll as an argument (the solid one or any other one built by *NestDoll* itself). Speaking intuitively, this constructor function does nothing other than *package up* the smaller nest doll it was given inside a "box" labelled *NestDoll*. One can thus obtain a nested doll either as the constant *SolidDoll* or by applying the *NestDoll* constructor some finite number of times to smaller nested dolls. Such a nesting will always be finitely deep, with the solid doll at the core.

A key idea in type theory is that *constructors are injective*. Two values of a given type built by different constructors, or by the same constructor with different arguments, are *always* different. So, for example, the solid doll is by definition unequal to any doll built by the *NestDoll* constructor; and a russian doll nested two levels deep (built by applying *NestDoll* to an argument representing a doll that is nested one level deep)is necessarily unequal to a russian doll one level deep (built by applying *NestDoll* to the solid doll).

Running this inequality idea in reverse, we can conclude that if two values of a given type are known to be equal, then for sure they were constructed by the same constructor taking the same arguments (if any). It turns out that knowing such a fact, rooted in the *injectivity of constructors* is often essential to completing proofs about programs using type theory. But more on this later.

7.10.3 Surjective

A function is said to be *surjective* if for every element, *t*, in the co-domain there is some element, *s* in the domain such that *(s,t)* is in the relation. That is, the range *range* of the function is its whole co-domain. Mathematically, a relation $R \subseteq S \times T$ is surjective if $\forall t \in T, \exists s \in S \mid (s,t) \in R$.

In the intuitive terms of high school algebra, a function involving *x* and *y* is surjective if for any given *y* value there is always some *x* that "leads to" that *y*. The *square* function on the real numbers is not surjective, because there is no *x* that when squared gets one to *y = -1*.

Exercise: Is the function, *f(x) = sin(x)*, from the real numbers (on the x-axis) to real numbers (on the y-axis) surjective? How might you phrase an informal but rigorous proof of your answer?

Exercise: Is the inverse of a surjective function always total? How would you "prove" this with a rigorous, step-by-step argument based on the definitions we've given here? Hint: It is almost always useful to start with definitions. What does it mean for a relation to be total? What does it mean for one relation to be the inverse of another? How can you connect these definitons to show for sure that your answer is right?

7.10.4 Bijective

A function is said to be *bjective* if it is also both injective and surjective. Such a function is also often called a *bijection*.

Take a moment to think about the implications of being a bijection. Consider a bijective relation, $R \subseteq S \times T$. *R* is total, so there is an *arrow* from every *s* in *S* to some *t* in *T*. *R* is injective, so no two arrows from any *s* in *s* ever hit the same *t* in *T*. An injection is one-to-one. So there is exactly one *t* in *T* hit by each *s* in *S*. But *R* is also surjective, so every *t* in *T* is hit by some arrow from *S*. Therefore, there has to be exactly one element in *t* for each element in *s*. So the sets are of the same size, and there is a one-to-one correspondence between their elements.

Now consider some *t* in *T*. It must be hit by exactly one arrow from *S*, so the *inverse* relation, R^{-1}, from *T* to *S*, must also single-valued (a function). Moreover, because *R* is surjective, every *t* in *T* is hit by some *s* in *S*, so the inverse relation is defined for every *t* in *T*. It, too, is total. Now every arrow from any *s* to some *t* leads back from that *t* to that *s*, so the inverse And it's also (and because *R* is total, there is such an arror for *every s* in *S*), the inverse relation is surjective (it covers all of *S*).

Exercise: Must the inverse of a bijection be one-to-one? Why or why not? Make a rigorous argument based on assumptions derived from our definitions.

Exercise: Must a bijective function be invertible? Make a rigorous argument.

Exercise: What is the inverse of the inverse of a bijective function, *R*. Prove it with a rigorous argument.

A bijection estabishes an invertible, one-to-one correspondence between elements of two sets. Bijections can only be established between sets of the same size. So if you want to prove that two sets are of the same size, it sufficies to show that one can define a bijection between the two sets. That is, one simply shows that there is some function that covers each element in each set with arrows connecting them, one-to-one in both directions.

Exercise: Prove that the number of non-negative integers (the cardinality of $\mathbb{N}$), is the same as the number of non-negative fractions (the cardinality of $\mathbb{Q}^+$).

Exercise: How many bijective relations are there between two sets of cardinality *k*? Hint: Pick a first element in the first set. There are *n* ways to map it to some element in the second set. Now for the second element in the first set, there are only *(n-1)* ways to pair it up with an element in the second set, as one cannot map it to the element chosen in the first step (the result would not be injective). Continue this line of reasoning until you get down to all elements having been mapped.

Exercise: How many bijections are there from a set, *S*, to itself? You can think of such a bijection as a simple kind of encryption. For example, if you map each of the *26* letters of the alphabet to some other letter, but in a way that is unambiguous (injective!), then you have a simple encryption mechanisms. How many ways can you encrypt a text that uses *26* letters in this way? Given a cyphertext, how would you recover the original plaintext?

Exercise: If you encrypt a text in this manner, using a bijection, *R* and then encrpty the resulting cyphertext using another one *T*, can you necessarily recover the plaintext? How? Is there a *single* bijection that would have accomplished the same encryption result? Would the inverse of that bijection effectively decrypt messages?

Exercise: Is the composition of any two bijections also a bijection? If so, can you express its inverse in terms of the inverses of the two component bijections?

Exercise: What is the *identity* bijection on the set of *26* letters?

Question: Are such bijections commutative? That is, you have two of them, say *R* and *T*, is the bijection that you get by applying *R* and then *T* the same as the bijection you get by applying *T* and then *R*? If your answer is *no*, prove it by giving a counterexample (e.g., involving bijections on a small set). If your answer is yes, make rigorous argument.

Programming exercise: Implement encryption and decryption schemes in Dafny using bijections over the *26* capital letters of the English alphabet.

Programming exercise: Implement a *compose* function in Dafny that takes two pure functions, *R* and *T*, each implementing a bijection between the set of capital letters and that returns a pure function that when applied has the effect of first applying *T* then applying *R*.

7.11 Properties of Relations

Functions are special cases of (single-valued) binary relations. The properties of being partial, total, injective, surjective, bijective are generally associated with *functions*, i.e., with relations that are already single-valued. Now we turn to properties of relations more generally.

7.11.1 Reflexive

Consider a binary relation on a set with itself. That is, the domain and the co-domain are the same sets. A relation that maps real numbers to real numbers is an example. It is a subset of $\mathbb{R} \times \mathbb{R}$. The *friends* relation on a social network site that associates people with people is another example.

Such a relation is said to be *reflexive* if it associates every element with itself. The equality relation (e.g., on real numbers) is the "canonical" example of a reflexive relation. It associates every number with itself and with no other number. The tuples of the equality relation on real numbers thus includes *(2.5, 2.5)* and (-3.0, -3.0)* but not *(2.5, -3.0)*.

In more mathematical terms, consider a set *S* and a bindary relation, *R*, on S*S, $R \subseteq S \times S$. *R* is reflexive, which we can write as *Reflexive(R)*, if and only if for every *e* in *S*, the tuple *(e,e)* is in R. Or to be rigorous about it, $Reflexive(R) \iff \forall e \in S, (e, e) \in R$.

Exercise: Is the function, *y = x*, reflexive? If every person loves themself, is the *loves* relation reflexive? Is the *less than or equals* relation reflexive? Hint: the tuples *(2,3)* and *(3,3)* are in this relation becaue *2* is less than or equal to *3*, and so is *3*, but *(4,3)* is not in this relation, because *4* isn't less than or equal to *3*. Is the less than relation reflexive?

7.11.2 Symmetric

A binary relation, *R*, on a set *S* is said to be *symmetric* if whenever the tuple *(x,y)* is in *R*, the tuple, *(y,x)* is in *R* as well. On Facebook, for example, if Joe is "friends" with "Tom" then "Tom" is necessarily also friends with "Joe." The Facebook friends relation is thus symmetric in this sense.

More formally, if *R* is a binary relation on a set *S*, i.e., given $R \subseteq S \times S$, then $Symmetric(R) \iff \forall (x, y) \in R, (y, x) \in R$.

Question: is the function *y = x* symmetric? How about the *square* function? In an electric circuit, if a conducting wire connects terminal *T* to terminal *Q*, it also connects terminal *Q* to terminal *T* in the sense that electricity doesn't care which way it flows over the wire. Is the *connects* relation in electronic circuits symmetric? If *A* is *near B* then *B* is *near A*. Is *nearness* symmetric? In the real work is the *has-crush-on* relation symmetric?

7.11.3 Transitive

Given a binary relation $R \subseteq S \times S$, *R* is said to be transitive if whenever *(x,y)* is in *R* and *(y,z)* is in *R*, then *(x,z)* is also in *R*. Formally, $Transitive(R) \iff forall(x, y)inR, \forall (y, z) \in R, (x, z) \in R$.

Exercise: Is equality transitive? That is, if *a = b* and *b = c* it is also necessarily the case that *a = c*? Answer: Sure, any sensible notion of an equality relation has this transitivity property.

Exercise: What about the property of being less than? If $a < b$ and $b < c$ is it necessarily the case that $a < c$? Answer: again, yes. The less than, as well as the less than or equal, and greater then, and the greater than or equal relations, are all transitive.

How about the *likes* relation amongst real people. If Harry likes Sally and Sally likes Bob does Harry necesarily like Bob, too? No, the human "likes" relation is definitely not transitive. (And this is the cause of many a tragedy.)

7.11.4 Equivalence

Finally (for now), a relation is said to be an *equivalence relation* if it is reflexive, transitive, and symmetric. Formally, we can write this property as a conjunction of the three individual properties: $Equivalence(R) \iff Symmetric(R) \wedge Reflexive(R) \wedge Transitive(R)$. Equality is the canonical example of an equivalence relation: it is reflexive ($x = x$), symmetric (if $x = y$ then $y = x$) and transitive (if $x = y$ and $y = z$ then $x = z$.).

An important property of equivalence relations is that they divide up a set into subsets of *equivalent* values. As an example, take the equivalence relation on people, *has same birthday as*. Clearly every person has the same birthday as him or herself; if Joe has the same birthday as Mary, then Mary has the same birthday as Joe; and if Tom has the same birthday as mary then Joe necessarily also has the same birthday as Tom. This relation thus divides the human population into 366 equivalence classes. Mathematicians usually use the notation $a \sim b$ to denote the concept that a is equivalent to b (under whatever equivalence relation is being considered).

7.12 Sequences

A sequence of elements is an ordered collection in which elements can appear zero or more times. In both mathematical writing and in Dafny, sequences are often denoted as lists of elements enclosed in square brackets. The same kinds of elisions (using elipses) can be used as shorthands in quasi-formal mathematical writing as with set notation. For example, in Dafny, a sequence *s := [1, 2, 3, 1]* is a sequence of integers, of length four, the elements of which can be referred to by subscripting. So *s[0]* is *1*, for example, as is *s[3]*.

While at first a sequence might seem like an entirely different kind of thing than a set, in reality a sequence of length, *n*, is best understood, and is formalized, as a binary relation. The domain of the relation is the sequence of natural numbers from *0* to *n-1*. These are the index values. The relation then associates each such index value with the value in that position in the sequence. So in reality, a sequence is a special case of a binary relation, and a binary relation is, as we've seen, just a special case of a set. So here we are, at the end of this chapter, closing the loop with where we started. We have seen that the concept of sets really is a fundamental concept, and a great deal of other machinery is then built as using special cases, including relations, maps, and sequences.

Tuples, too, are basically maps from indices to values. Whereas all the values in a sequence are necessarily of the same type, elements in a tuple can be of different types. Tuples also use the *.n* notation to apply projection functions to tuples. So, again, the value of, say, *("hello", 7).1* is *7* (of type *int*), while the value of *("hello", 7).0* is the string, "hello."

Sequences also support operations not supported for bare sets. These include sequence *concatenation* (addition, in which one sequence is appended to another to make a new sequence comprising the first one followed by the second. In Dafny, concatenation of sequences is done using the + operator. Dafny also has operations for accessing the individual elements of sequences, as well as subsequences. A given subsequence is obtained by taking a prefix of a suffix of a sequence. See the Dafny language summary for examples of these and other related operations on lists.

CHAPTER

EIGHT

8. BOOLEAN ALGEBRA

As a first stepping stone toward a deeper exploration of deductive logic, we explore the related notion of Boolean *algebra*. Boolean algebra is a mathematical framework for representing and reasoning about truth.

This algebra is akin to ordinary high school algebra, and as such, deals with values, operators, and the syntax and the evaluation of expressions involving values and operators. However, the values in Boolean algebra are limited to the two values in the set, $bool = \{0, 1\}$. They are often written instead as *false* and *true*, respectively. And rather than arithmetic operators such as numeric negation, addition, and subtraction, Boolean algebra defines a set of *Boolean operators*. They are typically given names such as *and*, *or*, and *not*, and they both operate on and yield Boolean values.

In this chapter, we first discuss Boolean algebra in programming, a setting with which the reader is already familar, baesd on a first course in programming. We then take a deeper look at the syntax and semantics of *expressions* in Boolean algebra. We do this by seeing how to use *inductive definitions* and *recursive functions* in the Dafny language to implement an *inductive data type* for representing Boolean expressions and a recursive *evaluation* function that when given any Boolean expression tells whether it is *true* or *false*.

8.1 Boolean Algebra in Dafny

All general-purpose programming languages support Boolean algebra. Dafny does so through its *bool* data type and the *operators* associated with it. Having taking a programming course, you will already have been exposed to all of the important ideas. In Dafny, as in many languages, the Boolean values are called *true* and *false* (rather than *1* and *0*).

The Boolean operators are also denoted not by words, such as *or* and *not* but by math-like operators. For example, *!* is the not operator and || is the *or* operator.

Here's a (useless) Dafny method that illustrates how Boolean values and operators can be used in Dafny. It presents a method, *BoolOps*, that takes a Boolean value and returns one. The commands within the method body illustrate the use of Boolean constant (literal) values and the unary and binary operators provided by the Dafny language.

```
method BoolOps(a: bool) returns (r: bool)
{
    var t: bool := true;    // explicit type declaration
    var f := false;         // type inferred automatically
    var not := !t;          // negation
    var conj := t && f;     // conjunction, short-circuit evaluation
    var disj := t || f;     // disjunction, short-circuit (sc) evaluation
    var impl := t ==> f;    // implication, right associative, sc from left
    var foll := t <== f;    // follows, left associative, sc from right
    var equv := t <==> t;   // iff, bi-implication
    return true;            // returning a Boolean value
 }
```

The first line assigns the Boolean constant, *true*, to a Boolean variable, *t*, that is explicitly declared to be of type,, *bool*. The second line assigns the Boolean constant, *false*, to *f*, and allows Dafny to infer that the type of f must be *bool*, based on the type of value being assigned to it. The third line illustrates the use of the *negation* operator, denoted as *!* in Dafny. Here the negation of *t* is assigned to the new Boolean variable, *not*. The next line illustrates the use of the Boolean *and*, or *conjunction* operator (*&&*). Next is the Boolean *or*, or *disjunction*, operator, (||). These should all be familiar.

Implication (==>) is a binary operator (taking two Boolean values) that is read as *implies* and that evaluates to false only when the first argument is true and the second one is false, and that evaluates to true otherwise. The *follows* operator (<==) swaps the order of the arguments, and evaluates to false if the first argument is false and the second is true, and evaluates to true otherwise. Finally, the *equivalence* operator evaluates to true if both arguments have the same Boolean value, and evaluates to false otherwise. These operators are especially useful in writing assertions in Dafny.

The last line returns the Boolean value true as the result of running this method. Other operations built into Dafny also return Boolean values. Arithmetic comparison operators, such as <, are examples. The less than operator, for example, takes two numerical arguments and returns true if the first is strictly less than the second, otherwise it returns false.

8.2 Boolean Values

Boolean algebra is an algebra, which is a set of values and of operations that take and return these values. The set of values in Boolean algebra, is just the set containing *0* and *1*.

$$bool = \{0, 1\}.$$

In English that expression just gave a name that we can use, *bool*, to the set containing the values, *0* and *1*. Although these values are written as if they were small natural numbers, you must think of them as elements of a different type. They aren't natural numbers but simply the two values in this other, Boolean, algebra. We could use different symbols to represent these values. In fact, they are often written instead as *false* (for *0*) and *true* (for *1*).The exact symbols we use to represent these values don't really matter. What really makes Boolean algebra what it is are the *operators* defined by Boolean algebra and how they behave.

8.3 Boolean Operators

An algebra, again, is a set of values of a particular kind and a set of operators involving that kind of value. Having introduced the set of two values of the Boolean type, let's turn to the *operations* of Boolean algebra.

8.3.1 Nullary, Unary, Binary, and n-Ary Operators

The operations of an algebra take zero or more values and return (or reduce to) values of the same kind. Boolean operators, for example, take zero or more Boolean values and reduce to Boolean values. An operator that takes no values (and nevertheless returns a value, as all operators do) is called a *constant*. Each value in the value set of an algebra can be though of as an operator that takes no values.

Such an operator is also called *nullary*. An operator that takes one value is called *unary*; one that takes two, *binary*, and in general, one that takes *n* arguments is called *n-ary* (pronounced "EN-airy").

Having already introduced the constant (*nullary*) values of Boolean algebra, each of the type we have called *bool*, we now introduce the types and behaviors the unary and binary Boolean operators, including each of those supported in Dafny.

8.3.2 The Unary Operators of Boolean Algebra

While there are two constants in Boolean algebra, each of type *bool*, there are four unary operators, each of type $bool \rightarrow bool$. This type, which contains an arrow, is a *function* type. It is the type of any function that first takes an argument of type *bool* then reduces to a value of type *bool*. It's easier to read, write, and say in math than in English. In math, the type would be prounced as "bool to bool."

There is more than one value of this function type. For example one such function takes any *bool* argument and always returns the other one. This function is of type "bool to bool", but it is not the same as the function that takes any bool argument and always returns the same value that it got. The type of each function is $bool \rightarrow bool$, but the function *values* are different.

In the programming field, the type of a function is given when it name, its arguments, and return values are declared. This part of a function definition is sometimes called the function *signature*, but it's just as well to think of it as decaring the function *type*. The *body* of the function, usually a sequence of commands enclosed in curly braces, describes its actual behavior, the particular function value associated with the given function name and type.

We know that there is more than one unary Boolean function. So how many are there? To specify the behavior of an operator completely, we have to define what result it returns for each possible combination of its argument values. A unary operator takes only one argument (of the given type). In Boolean algebra, a unary function can thus take one of only two possible values; and it can return only one of two possible result values. The answer to the question is just the number of ways that a function can *map* two argument values to two result values.

And the answer to this question is *four*. A function can map both *0* and *1* to *0*; both *0* and *1* to *1*; *0* to *0* and *1* to *1*; and *0* to *1* and *1* to *0*. There are no other possibilities. An easy-to-understand way to graphically represent the behavior of each of these operations is with a *truth table*.

The rows of a truth table depict all possible combinations of argument values in the columns to the left, and in the last column on the right a truth tables presents the corresponding resulting value. The column headers give names to the argument values and results column headers present expressions using mathematical logic notations that represent how the resulting values are computed.

Constant False

Here then is a truth table for what we will call the *constant_false* operator, which takes a Boolean argument, either *true* or *false*, and always returns *false.* In our truth tables, we use the symbols, *true* and *false*, instead of *1* and *0*, for consistency with the symbols that most programming languages, including Dafny, use for the Boolean constants.

P	$false$
true	false
false	false

Constant True

The *constant_true* operator always returns *true*.

P	$true$
true	true
false	true

Identity Function(s)

The Boolean *identity* function takes one Boolean value as an argument and returns that value, whichever it was.

P	P
true	true
false	false

As an aside we will note that *identity functions* taking any type of value are functions that always return exactly the value they took as an argument. What we want to say is that "for any type, *T*, and any value, *t* of that type, the identity function for type *T* applied to *t* always returns *t* itself. In mathematical logical notation, $\forall T : Type, \forall t : T, id_T(t) = t$. It's clearer in mathematical language than in English! Make sure that both make sense to you now. That is the end of our aside. Now back to Boolean algebra.

Negation

The Boolean negation, or *not*, operator, is the last of the four unary operators on Boolean values. It returns the value that it was *not* given as an argument. If given *true*, it evaluates to *false*, and if given *false*, to *true.*

The truth table makes this behavior clear. It also introduces the standard notation in mathematical logic for the negation operator, $\neg P$. This expression is pronounced, *not P*. It evaluates to *true* if *P* is false, and to *false* if *P* is *true*.

P	$\neg P$
true	false
false	true

8.3.3 Binary Boolean Operators

Now let's consider the binary operators of Boolean algebra. Each takes two Boolean arguments and returns a Boolean value as a result. The type of each such function is written $bool \rightarrow bool \rightarrow bool$, pronounced "bool to bool to bool." A truth table for a binary Boolean operator will have two columns for arguments, and one on the right for the result of applying the operator being defined to the argument values in the left two columns.

Because binary Boolean operators take two arguments, each with two possible values, there is a total of four possible combinations of argument values: *true* and *true*, *true* and *false*, *false* and *true*, and *false* and *false*. A truth table for a binary operator will thus have four rows.

The rightmost column of a truth table for an operator is really where the action is. It defines what result is returned for each combination of argument values. In a table with four rows, there will be four cells to fill in the final column. In a Boolean algebra there are two ways to fill each cell. And there are exactly *12^4 = 6* ways to do that. We can write them as *0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111*. There are thus exactly *16* total binary operators in Boolean algebra.

Mathematicians have given names to all *16*, but in practice we tend to use just a few of them. They are called *and*, *or*, and *not*. The rest can be expressed as combinations these operators. It is common in computer science also to use binary operations called *nand* (for *not and*), *xor* (for *exclusive or*) and *implies*. Here we present truth tables for each of the binary Boolean operators in Dafny.

And (conjunction)

The *and* operator in Boolean algebra takes two Boolean arguments and returns *true* when both arguments are *true*, and otherwise, *false*.

P	Q	$P \wedge Q$
true	true	true
true	false	false
false	true	false
false	false	false

Nand (not and)

The *nand* operator, short for *not and*, returns the opposite value from the *and* operator: *false* if both arguments are *true* and *true* otherwise.

P	Q	$P \uparrow Q$
true	true	false
true	false	true
false	true	true
false	false	true

As an aside, the *nand* operator is especially important for designers of digital logic circuits. The reason is that *every* binary Boolean operator can be simulated by composing *nand* operations in certain patterns. So if we have a billion tiny *nand* circuits (each with two electrical inputs and an output that is off only when both inputs are on), then all we have to do is connect all these little ciruits up in the right patterns to implement very complex Boolean functions. The capability to etch billions of tiny *nand* circuits in silicon and to connect them in complex ways is the heart of the computer revolution. Now back to Boolean algebra.

Or (disjunction)

The *or*, or *disjunction*, operator evaluates to *false* only if both arguments are *false*, and otherwise to *true*.

It's important to note that it evaluates to *true* if either one or both of its arguments are true. When a dad says to his child, "You can have a candy bar *or* a donut, *he likely doesn't mean *or* in the sense of *disjunction*. Otherwise the child well educated in logic would surely say, "Thank you, Dad, I'll greatly enjoy having both."

P	Q	$P \vee Q$
true	true	true
true	false	true
false	true	true
false	false	false

Xor (exclusive or)

What the dad most likely meant by *or* is what in Boolean algebra we call *exclusive or*, written as *xor*. It evalutes to true if either one, but *not both*, of its arguments is true, and to false otherwise.

P	Q	$P \oplus Q$
true	true	false
true	false	true
false	true	true
false	false	false

Nor (not or)

The *nor* operator returns the negation of what the *or* operator applied to the same arguments returns: *xor(b1,b2) = not(or(b1, b2))*. As an aside, like *nand*, the *nor* operator is *universal*, in the sense that it can be composed to with itself in different patterns to simulate the effects of any other binary Boolean operator.

P	Q	$P \downarrow Q$
true	true	false
true	false	false
false	true	false
false	false	true

Implies

The *implies* operator is used to express the idea that if one condition, a premise, is true, another one, the conclusion, must be. So this operator returns true when both arguments are true. If the first argument is false, this operator returns true. It returns false only in the case where the first argument is true and the second is not, because that violates the idea that if the first is true then the second must be.

P	Q	$P \rightarrow Q$
true	true	true
true	false	false
false	true	true
false	false	true

Follows

The *follows* operator reverses the sense of an implication. Rather than being understood to say that truth of the first argument should *lead to* the truth of the second, it says that the truth of the first should *follow from* the truth of the second.

P	Q	$P \leftarrow Q$
true	true	true
true	false	true
false	true	false
false	false	true

There are other binary Boolean operators. They even have names, though one rarely sees these names used in practice.

8.3.4 A Ternary Binary Operator

We can of course define Boolean operators of any arity. As just one example, we introduce a *ternary* (3-ary) Boolean operator. It takes three Boolean values as arguments and returns a Boolean result. It's type is thus ::*bool rightarrow bool rightarrow bool rightarrow bool*. We will call it *ifThenElse_{bool}*.

The way this operator works is that the value of the first argument determines which of the next two arguments values the function will return. If the first argument is *true* then the value of the whole expression is the value of the second argument, otherwise it is the value of the third. So, for example, *ifThenElse_{bool}(true, true, false)* evaluates to true, while *ifThenElse_{bool}(false, true, false)* is false.

It is sometimes helpful to write Boolean expressions involving *n-ary* operators for *n>1* using something other than function application (prefix) notation. So, rather than *and(true,false)*, with the operator in front of the arguments (*prefix* notation), we would typically write *true && false* to mean the same thing. We have first sed a symbol, *&&*, instead of the English word, *and* to name the operator of interest. We have also put the function name (now *&&*) *between* the arguments rather than in front of them. This is called *infix* notation.

With ternary and other operators, it can even make sense to break up the name of the operator and spread its parts across the whole expression. For example, instead of writing, *ifThenElse_{bool}(true, true, false)*, we could write it as *IF true THEN true ELSE false.* Here, the capitalized words all together represent the name of the function applied to the three Boolean arguments in the expression.

As an aside, when we use infix notation, we have to do some extra work, namely to specify the *order of operations*, so that when we write expressions, the meaning is unambiguous. We have to say which operators have higher and lower *precedence*, and whether operators are *left*, *right*, or not associative. In everyday arithmetic, for example, multiplication has higher precedence than addition, so the expression *3 + 4 * 5* is read as *3 + (4 * 5)* even though the + operator comes first in the expression.

Exercise: How many ternary Boolean operations are there? Hint: for an operator with *n* Boolean arguments there are 2^n combinations of input values. This means that there will be 2^n rows in its truth table, and so 2^n blanks to fill in with Boolean values in the right column. How many ways are there to fill in 2^n Boolean values? Express your answer in terms of n.

Exercise: Write down the truth table for our Boolean if-then-else operator.

8.4 Formal Languages: Syntax and Semantics

Any introduction to programming will have made it clear that there is an infinite set of Boolean expressions. For example, in Dafny, *true* is a Boolean expression; so are *false*, *true || false*, *(true || false) && (!false)*, and one could keep going on forever.

Boolean *expressions*, as we see here, are a different kind of thing than Boolean *values*. There are only two Boolean values, but there is an infinity of Boolean expressions. The connection is that each such expression has a corresponding Boolean truth value. For example, the expression, *(true || false) && (!false)* has the value, *true*.

The set of valid Boolean expressions is defined by the *syntax* of the Boolean expression language. The sequence of symbols, *(true || false) && (!false)*, is a valid expression in the language, for example, but *)(true false()||) false !&&* is not, just as the sequence of words, "Mary works long hours" is a valid sentence in the English language, but "long works hours Mary" isn't.

The syntax of a language defines the set of valid sentences in the language. The semantics of a language gives a meaning to each valid sentence in the language. In the case of Boolean expressions, the meaning given to each valid "sentence" (expression) is simply the Boolean value that that expression *reduces to*.

In the rest of this chapter, we use the case of Boolean expressions to introduce the concepts of the *syntax* and the *semantics* of *formal languages*. The syntax of a formal language precisely defines a set of *expressions* (sometimes called sentences or formulae). A *semantics* associates a *meaning*, in the form of a *value*, with each expression in the language.

8.5 The Syntax of Boolean Expressions: Inductive Definitions

As an example of syntax, the *true*, in the statement, *var b := true;* is a valid expression in the language of Boolean expressions, as defined by the *syntaxt* of this language. The semantics of the language associates the Boolean *value*, *true*, with this expression.

You probably just noticed that we used the same symbol, *true*, for both an expression and a value, blurring the distinction between expressions and values. Expressions that directly represent values are called *literal expressions*. Many languages use the usual name for a value as a literal expression, and the semantics of the language then associate each such expression with its corresponding value.

In the semantics of practical formal languages, literal expressions are assigned the values that they name. So the *expression*, *true*, means the *value*, *true*, for example. Similarly, when *3* appears on the right side of an assignment/update statement, such as in *x := 3*, it is an *expression*, a literal expression, that when *evaluated* is taken to *mean* the natural number (that we usually represent as) *3*.

As computer scientists interested in languages and meaning, we can make these concepts of syntax and semantics not only precisely clear but also *runnable*. So let's get started.

8.5.1 The Syntax and Semantics of *Simplified* Boolean Expression Language

We start by considering a simplified language of Boolean expressions: one with only two literal expressions. To make it clear that they are not Boolean values but expressions, we will call them not *true* and *false* but *bTrue* and *bFalse*.

Syntax

We can represent the syntax of this language in Dafny using what we call an *inductive data type definition.* A data type defines a set of values. So what we need to define is a data type whose values are all and only the valid expressions in the language. The data type defines the *syntax* of the language.

In the current case, we need a type with only two values, each one of them representing a valid expression in our language. Here's how we'd write it in Dafny.

```
datatype Bexp =
     bTrue |
     bFalse
```

The definition starts with the *datatype* keyword, followed by the name of the type being defined (*Bexp*, short for Boolean expression) then an equals sign, and finally a list of *constructors* separated by vertical bar characters. The constructors define the ways in which the values of the type (or *terms*) can be created. Each constructor has a and can take optional parameters. Here the names are *bTrue* and *bFalse* and neither takes any parameters.

The only values of an inductive type are those that can be built using the provided constructors. So the language that we have specified thus far has only two values, which we take to be the valid expressions in the language we are specifying, namely *bTrue* and *bFalse*. That is all there is to specifying the *syntax* of our simplified language of Boolean expressions.

Semantics

To give a preview of what is coming, we now specify a semantics for this language. Speaking informally, we want to associate, to each of the expressions, a correponding meaning in the form of a Boolean value. We do this by defining a *function* that takes an expression (a value of type *bExp*) as an argument and that returns the Boolean *value* that the semantics defines as the meaning of that expression. Here, we want a function that returns Dafny's Boolean value *true* for the expression, *bTrue*, and the Boolean value *false* for *bFalse*.

Here's how we can write this function in Dafny.

```
function method bEval(e: bExp): bool
{
  match e
  {
```

```
      case bTrue => true
      case bFalse => false
  }
}
```

As a shorhand for *Boolean semantic evaluator* we call it *bEval*. It takes an expression (a value of type, *bExp*) and returns a Boolean value. The function implementation uses an important construct that is probably new to most readers: a *match* expression. What a match expression does is to: first determine how a value of an inductive type was buit, namely what constructor was used and what arguments were provided (if any) and then to compute a result for the case at hand.

The match expression starts with the match keyword followed by the variable naming the value being matched. Then within curly braces there is a *case* for each constructor for the type of that value. There are two constructors for the type, *bExp*, so there are two cases. Each case starts with the *case* keyword, then the name of a constructor followed by an argument list if the construtor took parameters. Neither constructor took any parameters, so there is no need to deal with parameters here. The left side thus determines how the value was constructed. Each case has an arrow, =>, that is followed by an expression that when evaluated yields the result *for that case*.

The code here can thus be read as saying, first look at the given expression, then determine if it was *bTrue* or *bFalse*. In the first case, return *true*. In the second case, return *false*. That is all there is to defining a semantics for this simple language.

8.5.2 The Syntax of a Complete Boolean Expression Language

The real language of Boolean expressions has many more than two valid expressions, of course. In Dafny's Boolean expression sub-language, for example, one can write not only the literal expressions, *true* and *false*, but also expressions such as *(true || false) && (not false)*.

There is an infinity of such expressions, because given any one or two valid expressions (starting with *true* and *false*) we can always build a bigger expression by combing the two given ones with a Boolean operator. No matter how complex expressions *P* and *Q* are, we can, for example, always form the even more complex expressions, *!P*, *P && Q*, and *P || Q*, among others.

How can we extend the syntax of our simplified language so that it specifies the infinity set of well formed expressions in the language of Boolean expressions? The answer is that we need to add some more cosntructors. In particular, for each Boolean operator (including *and, or*, and *not*), we need a a constructor that takes the right number of smaller expressions as arguments and the builds the right larger expression.

For example, if *P* and *Q* are arbitrary "smaller" expressions, we need a consructor to build the expression *P and Q*, a constructor to build the expression, *P or Q*, and one that can build the expressions *not P* and *not Q*. Here then is the induction: some constructors of the *bExp* type will take values of the very type we're defining as parameters. And because we've defined some values as constants, we have some expressions to get started with. Here's how we'd write the code in Dafny.

```
datatype bExp =
     bTrue |
     bFalse |
     bNot (e: bExp) |
     bAnd (e1: bExp, e2: bExp) |
     bOr (e1: bExp, e2: bExp)
```

We've added three new constructors: one corresponding to each of the *operator* in Boolean algebra (to keep things simple, we're dealing with only three of them here). We have named each constructor in a way that makes the connection to the corresponding operator clear.

We also see that these new constructors take parameters. The *bNot* constructor takes a "smaller" expression, *e*, and builds/returns the expression, *bNot e*, which we will interpret as *not e*, or, as we'd write it in Dafny, *!e*. Similarly,

given expressions, *e1* and *e2*, the *bAnd* and *bOr* operators construct the expressions *bAnd(e1,e2)* and *bOr(e1,e2)*, respectively, representing *e1 and e2* and *e1 or e2*, respectively, or, in Dafny syntax, *e1 && e2* and *e1 || e2*.

An expression in our *bExp* language for the Dafny expression *(true || false) and (not false))* would be written as *bAnd((bOr (bTrue, bFalse)), (bNot bFalse))*. Writing complex expressions like this in a single line of code can get awkward, to we could also structure the code like this:

```
var T: bExp := bTrue;
var F:      := bFalse;
var P:      := bOr ( T,  F );
var Q       := bNot ( F );
var R       := bAnd ( P, Q );
```

8.6 The Semantics of Boolean Expressions: Recursive Evaluation

The remaining question, then, is how to give meanings to each of the expressions in the infinite set of expressions that can be built by finite numbers of applications of the constructor of our extended *bExp* type? When we had only two values in the type, it was easy to write a function that returned the right meaning-value for each of the two cases. We can't possibly write a separate case, though, for each of an infinite number of expressions. The solution lies again in the realm of recursive functions.

Such a function will simply do mechanically what you the reader would do if presented with a complex Boolean expression to evaluate. You first figure out what operator was applied to what smaller expression or expressions. You then evaluate those expressions to get values for them. And finally you apply the Boolean operator to those values to get a result.

Take the expression, *(true || false) and (not false))*, which in our language is expressed by the term, *bAnd((bOr (bTrue, bFalse)), (bNot bFalse))*. First we identify the *constructor* that was used to build the expression In this case it's the constructor corresponding to the *and* operator: *&&* in the Dafny expression and the *bAnd* in our own expression language. What we then do depends on what case has occured.

In the case at hand, we are looking at the constructor for the *and* operator. It took two smaller expressions as arguments. To enable the precise expression of the return result, ew given temporary names to the argument values that were passed to the constructor. We can call them *e1* and *e2*, for example. sub-expressions that the operator was applied to.

In this case, *e1* would be *(true || false)* and *e2* would be *(not false)*. To compute the value of the whole expression, we then obtain Boolean values for each of *e1* and *e2* and then combine them using the Boolean *and* operator.

The secret is that we get the values for *e1* and *e2* by the very same means: recursively! Within the evaluation of the overall Boolean expression, we thus recursively evaluate the subexpressions. Let's work through the recursive evaluation of *e1*. It was built using the *bOr* constructor. That constructor took two arguments, and they were, in this instance, the literal expressions, *bTrue* and *bFalse*. To obtain an overall result, we recursively evaluate each of these expressions and then combine the result using the Boolean *or* operator. Let's look at the recursive evaluation of the *bTrue* expression. It just evaluates to the Boolean value, *true* with no further recursion, so we're done with that. The *bFalse* evaluates to *false*. These two values are then combined using *or* resulting in a value of *true* for *eq*. A similarly recursive process produces the value, *true*, for *e2*. (Reason through the details yourself!) And finally the two Boolean values, *true* and *true* are combined using Boolean *and*, and a value for the overall expression is thus computed and returned.

Here's the Dafny code.

```
function method bEval(e: bExp): (r: bool)
{
    match e
    {
        case bTrue => true
```

```
        case bFase => false
        case bNot(e: bExp) => !bEval(e)
        case bAnd(e1, e2) => bEval(e1) && bEval(e2)
        case bOrEe1, e2) =>  bEval(e1) || bEval(e2)
    }
}
```

This code extends our simpler example by adding three cases, one for each of the new constructor. These constructors took smaller expression values as arguments, so the corresponding cases have used parameter lists to temporarily give names (*e1*, *e2*, etc.) to the arguments that were given when the constructor was orginally used. These names are then used to write the expressions on the right sides of the arrows, to compute the final results.

These result-computing expressions use recursive evalation of the constitute subexpressions to obtain their meanings (actual Boolean values in Dafny) which they then combine using actual Dafny Boolean operators to produce final results.

The meaning (Boolean value) of any of the infinite number of Boolean expressions in the Boolean expression language defined by our syntax (or *grammar*) can be found by a simple application of our *bEval* function. To compute the value of *R*, above, for example, we just run *bEval(R)*. For this *R*, this function will without any doubt return the intended result, *true*.

8.7 The Syntax and Semantics of Programming Languages

Syntax defines legal expressions. Semantics give each legal expression an associated meaning. The meanings of Boolean expressions are Boolean values. Using exactly the same ideas used here for Boolean expressions we could not only specify but compute with the syntax semantics of a language of arithmetic expressions.

Indeed, the same ideas apply to programming language. A programming language has a syntax. It defines the set of valid "programs" in that language. A programming language also has a semantics, It specifies what each such program means. However, th meaning of a program is not captured in a single value. Rather, it is expressed ina relation that explains how running the programs transforms any pre-execution state that satisfies the program preconditions into a post-execution state.

CHAPTER

NINE

9. PROPOSITIONAL LOGIC

This chapter first introduces logic, in general, and the introduces propositional logic, in particular. Proposition logic is a simple but useful logic that is very closely related to Boolean algebra.

9.1 Basic Terminology

Here is a proposition: "Tom's mother is Mary." A proposition asserts that a particular *state of affairs* holds in some particular *domain of discourse*. The domain in this case would be some family unit; and the state of affairs that is asserted to prevail is that Mary is Tom's Mom.

Whether such a proposition can be *judged* to be *true* is another matter. If the domain of discourse (or just *domain*) were that of a family in which there really are people, Tom and Mary, and in that family unity Mary really is the mother of Tom, then this proposition could be judged to be true. However, if such a state of affairs did not hold in that family unity, then the proposition would still be a fine proposition, but it could not be judged to be true (and indeed could be judged to be false).

In place of the phrase "domain of discourse" we could also use the word, "world." In general, the truth of a proposition depends on the world in which it is evaluated. There are some proposition that are true in every world, e.g., "zero equals zero;" some that are not true in any world, e.g., "zero equals one;" and many where the truth of the proposition depends on the world in which it is evaluated, e.g., "Mary is Tom's mother."

9.2 Propositional and Predicate Logic

The proposition, *Tom's mother is Mary*, is simple. It could even be represented as a single variable, let's call it *M*. In what we call propositional logic, we generally represent propositions as variables in this manner. Similarly, a logical variable, *F*, could represent the proposition, *Tom's father is Ernst*. We could then *construct* a larger proposition by composing these two propositions into a larger one under the logical connective called *and*. The result would be the proposition, *Toms' mother is Mary **and* Tom's father is Ernst*. We could of course write this more concisely as *M and F*, or, in a more mathematical notation, $M \wedge F$).

Now we ask, what is the truth value of this larger proposition? To determine the answer, we conjoin the truth values of the constituent propositions. The meaning of the larger proposition is determined by (1) the meanings of its smaller parts, and (2) the logical connective that composes them into a larger proposition. Such a logic of simple propositions and their compositions using connectives such as *and*, *or*, and *not* is called *propositional logic*.

By contrast, the proposition, "every person has a mother" (or to put it more formally, $\forall p \in Persons, \exists m \in Persons, motherOf(p, m)$), belongs to a richer logic. Here, *motherOf(p, m)* is a *predicate on two values*. It stands for the family of propositions, *the mother of p is m*, where *p* and *m* are variables that range over the set of people in the given domain of discourse. The overall proposition thus asserts that, for *every* person, *p* in the domain, there is a person, *m*, such that *m* is the mother of *p*.

The *motherOf(p,m)* construct is a *parameterized proposition*, which, again, we call a *predicate*. You can think of it as a function that takes two values, *p* and *m*, and that returns a proposition *about those particular values*. A predicate thus represents not a just one proposition but a whole *family* of propositions, one for each pair of parameter values. Any particular proposition of this form might be true or false depending on the domain of discourse. If *m* really is the mother of *p* (in the assumed domain), then *motherOf(p,m)* can be judged to be true (for that domain), and otherwise not.

Another way to look at a predicate is that it *picks out* a subset of *(p,m)* pairs, namely all and only those for which *motherOf(p,m)* is true. A predicate thus specifies a relation, here a binary relation, namely the *motherOf* relation on the set of people in the domain of discourse.

This richer logic, called *predicate logic*, (1) allows variables, such as *p* and *m*, to range over sets of objects (rather that just over Boolean values), (2) supports the expression predicates involving elements of such sets, and (3) provides botg universal and existential quantifiers (*for all* and *there exists*, respectively). As we will see in a later chapter, a predicate logic also allows the definition and use of functions taking arguments in the domain to identify other objects in the domain. So, for example, a function, *theMotherOf*, might be used to identify *Mary* as *theMotherOf(Tom)*. Note that when a function is applied to domain values, the result is another domain value, whereas when a predicate is applied to domain values, the result is a proposition about those value.

Predicate logic is the logic of everyday mathematics and computer science. It is, among other things, the logic Dafny provides for writing specifications. As an example, consider our specification of what it means for a function, *R*, with domain, *D*, and codomain, *C*, to be surjective: $\forall c \in C, \exists d \in D, (d, c) \in R$. In Dafny, we would (and did) write this as, *forall c :: c in codom() ==> exists d :: d in dom() && (d,c) in rel()*. Dafny is thus a specification and verification system based on *predicate logic*. We've been using it all along!

One of the main goals of this course up to now has been to get you reading, writing, and seeing the utility of predicate logic. Far from being an irrelevancy, it is one of the pillars of computer science. It is a fundamental tool for specifying and reasoning about software. It is also central to artificial intelligence (AI), to combinatorial optimization (e.g., for finding good travel routes), in the analysis of algorithms, in digital circuit design, and in many other areas of computer science, not to mention mathematics and mathematical fields such as economics.

Going forward, one of our main goals is to understand predicate logic in greater depth, including its syntax (what kinds of expressions can you write in predicate logic?) its semantics (when are expressions in predicate logic *true?*), and how to show that given expressions are true.

In this chapter, which beings Part II of this set of notes, we start our exploration of predicate logic and proof by first exploring the simpler case of *propositional* logic. To begin, in the next section, we address the basic question, what is *a logic*, in the first place?

9.3 What is a Logic?

A logic is (1) a *formal language* of *propositions* along with (2) principles for reasoning about whether any given proposition can be judged to be *true* or not. A logic has a *syntax*, which is a set of mathematically (formally) specified rules that precisely define the set of well formed propositions (or *well formed formulae, or wffs*) in the language. A logic also has a *semantics*, which is a set of formal rules for reasoning about whether a given proposition can be judged to be true or not.

In the last chapter, on Boolean algebra, we already saw what amounts to a simplified version of propositional logic, with both a syntax and a semantics! The syntax of our Boolean expression language is given by the inductive *bExp* type. It provides a set of constructors, which are just the rules for building valid expressions, with an implicit assumption that the valid expressions in the language are all and only those that can be built using the provided constructors. The syntax is compositional, in that smaller expressions can be composed into ever larger ones, up to arbitrarily large (but always still finite) sizes.

The semantics of the simplified logic is then defined by a *semantic evaluation* function, that takes *any* valid expression in the language as an argument and that returns a Boolean value indicating whether the given expression is (can be

judged to be) true or not. The semantics is also compositional in that the truth of a composed proposition is defined recursively in terms of (1) the truth values of its constituent propositions, and (2) the meaning of the connector that was used to compose them. The recursive structure of semantic evaluation exactly mirrors the inductive definition of the syntax.

9.4 Propositional Logic

We now introduce propositional logic. The syntax of propositional logic is basically that of our Boolean expression language with the crucial addition of propositional *variable expressions*. Examples of variable expressions include *M* and *F* in our example at the start of this chapter. So, for example, in addition to being able to write expressions such as *pAnd(pTrue,pFalse)*, we can write *pAnd(pTrue,F)* and *POr(M,F)*.

As for semantics, propositional variables take Boolean values. To evaluate a variable expression, we just look up its Boolean value and then proceed as with Boolean expression evaluation in the last chapter.

The one complication, then, is that, to evaluate a proposition (which in general includes variables), our semantic evaluation function needs to have a way to look up the Boolean value of each variable appearing in the expression being evaluated. Our semantic evaluator need a *map* from variables to values. Logicians call such a variable-to-value map an *interpretation*. Programming language designers call such a map an *environment*. To evaluate a variable expression, the evaluator will just look up its value in the given intepretation and will otherwise proceed as in the last chapter.

9.5 Inductive Definitions: The Syntax of Propositional Logic

A logic provides a *formal language* in which propositions (truth statements) are expressed. By a formal language, we mean a (usually infinite) set of valid expressions in the language. For example, the language of Boolean expressions includes the expression *true and false* but not *and or true not*.

When the set of valid expressions in a language is infinite in size, it becomes impossible to define the language by simply listing all valid expressions. Instead, the set of valid expressions is usually defined *inductively* by a *grammar*. A grammar defines a set of elementary expressions along with a set of rules for forming ever larger expressions from ones already known to be in the language. We also call the grammar for a formal language its *syntax*.

The syntax of proposition logic is very simple. First, (with details that vary among presentations of propositional logic), it accepts two *literal values*, usually called *true* and *false*, as expressions. Here we will call these values *pFalse* and *pTrue* to emphasize that these are *expressions* that we will eventually *interpret* as having particular Boolean values (namely *false* and *true*, respectively).

Second, propositional logic assumes an infinite set of *propositional variables*, each represents a proposition, and each on its own a valid expression. For example, the variable, *X*, might represent the basic proposition, "It is raining outside," and *Y*, that "The streets are wet." Such variables should be understood as being equated with basic propositions. Instead of the identifier, *X*, one might just as well have used the identifier, *it_is_raining_outside*, and for *Y*, the identifier, *the_streets_are_wet*.

Finally, in addition to literal values and propositional variables, propositional logic provides the basic Boolean connectives to build larger propositions from smaller ones. So, for example, *X and Y*, *X or Y*, and *not X* are propositions constructed by the use of these *logical connectives.* So is *(X or Y) and (not X)*. (Note that here we have included parentheses to indicate grouping. We will gloss over the parentheses as part of the syntax of propositional logic.)

We have thus defined the entire syntax of propositional logic. We can be more precise about the grammar, or syntax, of the language by giving a more formal set of rules for forming expressions.

```
Expr       := Literal | Variable | Compound
Literal    := pFalse | pTrue
```

```
Variable   := X | Y | Z | ...
Compound   := Not Expr | And Expr Expr | Or Expr Expr
```

This kind of specification of a grammar, or syntax, is said to be in *Backus-Naur Form" or BNF, after the names of two researchers who were instrumental in developing the theory of programming languages. (Every programming language has such a grammar.)

This particular BNF grammar reads as follows. A legal expression is either a literal expression, a variable expression, or a compound expression. A literal expression, in turn, is either *pTrue* or *pFalse*. (Recall that these are not Boolean values but Boolean *expressions* that *evaluate* to Boolean values.) A variable expression is X, Y, Z, or any another variable letter one might wish to employ. Finally, if one already has an expression or two, one can form a larger expression by putting the *Not* connective in front of one, or an *And* or *Or* connective in front of two expressions. That is the entire grammar of propositional logic. (Some presentations of propositional logic leave out the literal expressions, *pTrue* and *pFalse*.)

Here's the corresponding completely formal code in Dafny. First, to represent *variables*, we define a datatype called *propVar*, with a single constructor called *mkPropVar*, that takes a single argument, *name*, of type *string*. Examples of variable objects of this type thus include *mkPropVar("M")* and *mkPropVar("F")*. Two variables of this type are equal if and only if their string arguments are equal.

```
datatype propVar = mkPropVar(name: string)
```

With that, we can now give a Dafny specification of the syntax of our version of propositional logic. It's exactly the same as the syntax of Boolean expressions from the last chapter but for the addition of one new kind of expression, a *variable expression*, which is built using the *pVar* constructor applied to a *variable* (that is, a value of type *propVar*).

```
datatype prop =
   pTrue |
   pFalse |
   pVar (v: propVar) |
   pNot (e: prop) |
   pAnd (e1: prop, e2: prop) |
   pOr (e1: prop, e2: prop) |
   pImpl (e1: prop, e2: prop)
```

This kind of definition is what we call an *inductive definition*. The set of legal expressions is defined in part in terms of expressions! It's like recursion. What makes it work is that one starts with some non-recursive *base* values, and then the inductive rules allow them to be put together into ever larger expressions. Thinking in reverse, one can always take a large expression and break it into parts, using recursion until base cases are reached.

Note that we distinguish *variables* (values of type *propVar*) from *variable expressions* (values of type *prop*). This approach makes it easy to represent an interpretation as a map from variables (of type *propVar*) to Boolean values.

9.6 Semantics of Propositional Logic

Second, a logic defines a of what is required for a proposition to be judged true. This definition constitutes what we call the *semantics* of the language. The semantics of a logic given *meaning* to what are otherwise abstract mathematical expressions; and do so in particular by explaining when a given proposition is true or not true.

The semantics of propositional logic are simple. They just generalize the semantics of our Boolean expression language by also supporting the evaluation of propositional variable expressions.

The literal expressions, *pTrue* and *pFalse* still evaluate to Boolean *true* and *false*, respectively. A variable can have either the value, *true* or the value, *false*. To evaluate the value of any particular variable expression, one obtains the underlying variable and looks up its Boolean values in a given *interpretation*. Recall that an interpretation is just a *map* (or *function*) from variables to Boolean values. Finally, an an expression of the form *pAnd e1 e2*, *pOr e1 e2*, or *pNot*

e are evaluated just as they were in the last chapter, by recursively evaluating the sub-expressions and combining the values using the Boolean operator corresponding to the constructor that was used to build the compound expression. Evaluation of a larger expression is done by recursively evaluating smaller expressions until the base cases of *pTrue* and *pFalse* are reached.

Here's the Dafny code for semantic evaluation of any proposition (an expression object of type *prop*) in our propositional logic language.

```
function method pEval(e: prop, i: pInterpretation): (r: bool)
     requires forall v :: v in getVarsInProp(e) ==> v in i
 {
     match e
     {
          case pTrue => true
          case pFalse => false
          case pVar(v: propVar) => pVarValue(v,i)
          case pNot(e1: prop) => !pEval(e1,i)
          case pAnd(e1, e2) => pEval(e1,i) && pEval(e2, i)
          case pOr(e1, e2) =>  pEval(e1, i) || pEval(e2, i)
          case pImpl(e1, e2) => pEval(e1, i) ==> pEval(e2, i)
     }
 }
```

Our semantic evaluation function is called *pEval*. It takes a proposition expression, e, and an interpration, *i*, which is just a map from variables (of type *propVar*) to Boolean values, i.e., a value of type *map<propVar,bool>*. The precondition is stated using an auxiliary function we've define; and overall it simply requires that there be a value defined in the map for any variable that appears in the given expression, *e*. Finally, the evaluation procedure is just as it was for our language of Boolean algebra, but now there is one more rule: to evaluate a variable expression (built using the *propVar* constructor), we just look up its value in the given map (interpretation).

Exercise: Write a valid proposition using our Dafny implementation to represent the assertion that *either it is not raining outside or the streets are wet.* Use only one logical connective.

Exercise: Extend the syntax above to include an *implies* connective and express the proposition from the previous exercise using it. (Okay, the code already implements it, so this exercise is obsolete.)

9.7 Inference Rules for Propositional Logic

Finally, a logic provides a set of *inference rules* for deriving new propositions (conclusions) from given propositions (premises) in ways that guarantee that if the premises are true, the conclusions will be, too. The crucial characteristic of inference rules is that although they are guarantee to *preserve meaning* (in the form of truthfulness of propositions), they work entirely at the level of syntax.

Each such rule basically says, "if you have a set of premises with certain syntactic structures, then you can combine them in ways to derive new propositions with absolute certainty that, if the premises are true, the conclusion will be, too. Inference rules are thus rules for transforming *syntax* in ways that are *semantically sound*. They allow one to derive *meaningful* new conclusions without ever having to think about meaning at all.

These ideas bring us to the concept of *proofs* in deductive logic. If one is given a proposition that is not yet known to be true or not, and a set of premises known or assumed to be true, a proof is simply a set of applications of availabile inference rules in a way that, step by step, connects the premises *syntactically* to the conclusion.

A key property of such a proof is that it can be checked mechanically, without any consideration of *semantics* (meaning) to determine if it is a valid proof or not. It is a simple matter at each step to check whether a given inference rule was applied correctly to convert one collection of propositions into another, and thus to check whether *chains* of inference rules properly connect premises to conclusions.

For example, a simple inference rule called *modus ponens* states that if *P* and *Q* are propositions and if one has as premises that (1) *P* is true*, and (2) *if P is true then Q is true*, then one can deduce that *Q is true*. This rule is applicable *no matter what* the propositions *P* and *Q* are. It thus encodes a general rule of sound reasoning.

A logic enables *semantically sound* "reasoning" by way of syntactic transformations alone. And a wonderful thing about syntax is that it is relatively easy to mechanize with software. What this means is that we can implement systems that can reasoning *meaningfully* based on syntactic transformation rules alone.

Note: Modern logic initially developed by Frege as a " formula language for pure though,t modeled on that of arithmetic," and later elaborated by Russel, Peano, and others as a language in which, in turn, to establish completely formal foundations for mathematics.

9.8 Using Logic in Practice

To use a logic for practical purposes, one must (1) understand how to represent states of affairs in the domain of discourse of interest as expressions in the logical language of the logic, and (2) havee some means of evaluating the truth values of the resulting expressions. In Dafny, one must understand the logical language in which assertions and related constructs (such as pre- and post-conditions) are written.

In many cases–the magic of an automated verifier such as Dafny–a programmer can rely on Dafny to evaluate truth values of assertions automatically. When Dafny is unable to verify the truth of a claim, however, the programmer will also have to understand something about the way that truth is ascertained in the logic, so as to be able to provide Dafny with the help it might need to be able to complete its verification task.

In this chapter, we take a major step toward understanding logic and proofs by introducing the language *propositional logic* and a means of evaluating the truth of any sentence in the language. The language is closely related to the language of Boolean expressions introduced in the last chapter. The main syntactic difference is that we add a notion of *propositional variables*. We will defined the semantics of this language by introducing the concept of an *interpration*, which specifies a Boolean truth value for each such variable. We will then evaluate the truth value of an expression *given an interpration for the proposition variables in that expression* by replacing each of the variables with its corresponding Boolean value and then using our Boolean expression evaluator to determing the truth value of the expression.

We will also note that this formulation gives rise to an important new set of logical problems. Given an expression, does there exist an interpretation that makes that expression evaluate to true? Do all interpretations make it value to true? Can it be there there are no interpretations that make a given expression evaluate to true? And, finally, are there *efficient* algorithms for *deciding* whether or not the answer to any such question is yes or no.

9.9 Implementing Propositional Logic

The rest of this chapter illustrates and further develops these ideas using Boolean algebra, and a language of Boolean expressions, as a case study in precise definition of the syntax (expression structure) and semantics (expression evaluation) of a simple formal language: of Boolean expressions containing Boolean variables.

To illustrate the potential utility of this language and its semantics we will define three related *decision problems*. A decision problem is a *kind* of problem for which there is an algorithm that can solve any instance of the problem. The three decision problems we will study start with a Boolean expression, one that can contain variables, and ask where there is an assignment of *true* and *false* values to the variables in the expression to make the overall expression evaluate to *true*.

Here's an example. Suppose you're given the Boolean expression, $(P \vee Q) \wedge (\neg R)$. The top-level operator is *and*. The whole expression thus evaluates to *true* if and only if both subexpressions do: $(P \vee Q)$ and $\wedge(\neg R)$, respectively. The first, $(P \vee Q)$, evaluates to *true* if either of the variables, *P* and *Q*, are set to true. The second evaluates to true if

and only if the variable *R* is false. There are thus settings of the variables that make the formula true. In each of them, *R* is *false*, and either or both of *P* and *Q* are set to true.

Given a Boolean expression with variables, an *interpretation* for that expression is a binding of the variables in that expression to corresponding Boolean values. A Boolean expression with no variables is like a proposition: it is true or false on its own. An expression with one or more variables will be true or false depending on how the variables are used in the expression.

An interpretation that makes such a formula true is called a *model*. The problem of finding a model is called, naturally enough, the model finding problem, and the problem of finding *all* models that make a Boolean expression true, the *model enumeration* or *model counting* problem.

The first major *decision problem* that we identify is, for any given Boolean expression, to determine whether it is *satisfiable*. That is, is there at least one interpretation (assignment of truth values to the variables in the expression that makes the expression evaluate to *true*? We saw, for example, that the expression, $(P \vee Q) \wedge (\neg R)$ is satifiable, and, moreover, that $\{(P, true), (Q, false), (R, false)\}$ is a (one of three) interpretations that makes the expression true.

Such an interpretation is called a *model*. The problem of finding a model (if there is one), and thereby showing that an expression is satisfiable, is naturally enough called the* model finding* problem.

A second problem is to determine whether a Boolean expression is *valid*. An expression is valid if *every* interpretation makes the expression true. For example, the Boolean expression $P \vee \neg P$ is always true. If *P* is set to true, the formula becomes $true \vee false$. If *P* is set to false, the formula is then $true \vee false$. Those are the only two interpretations and under either of them, the resulting expression evaluates to true.

A third related problem is to determine whether a Boolean expression is it *unsatisfiable*? This case occurs when there is *no* combination of variable values makes the expression true. The expression $P \wedge \neg P$ is unsatisfiable, for example. There is no value of P (either *true* or *false*) that makes the resulting formula true.

These decision problems are all solvable. There are algorithms that in a finite number of steps can determine answers to all of them. In the worst case, one need only look at all possible combinations of true and false values for each of the (finite number of) variables in an expression. If there are *n* variables, that is at most 2^n combinations of such values. Checking the value of an expression for each of these interpretations will determine whether it's satisfiable, unsatisfiable, or valid. In this chapter, we will see how these ideas can be translated into runnable code.

The much more interesting question is whether there is a fundamentally more efficient approach than checking all possible interpretations: an approach with a cost that increases *exponentially* in the number of variables in an expression. This is the greatest open question in all of computer science, and one of the greatest open questions in all of mathematics.

So let's see how it all works. The rest of this chapter first defines a *syntax* for Boolean expressions. Then it defines a *semantics* in the form of a procedure for *evaluating* any given Boolean expression given a corresponding *interpretation*, i.e., a mapping from variables in the expression to corresponding Boolean values. Next we define a procedure that, for any given set of Boolean variables, computes and returns a list of *all* interpretations. We also define a procedure that, given any Boolean expression returns the set of variables in the expression. For ths set we calculate the set of all interpretations. Finally, by evaluating the expression on each such interpretation, we decide whether the expression is satisfiable, unsatisfiable, or valid.

Along the way, we will meet *inductive definitions* as a fundamental approach to concisely specifying languages with a potentially infinite number of expressions, and the *match* expression for dealing with values of inductively defined types. We will also see uses of several of Dafny's built-in abstract data types, including sets, sequences, and maps. So let's get going.

9.9.1 Syntax

Any basic introduction to programming will have made it clear that there is an infinite set of Boolean expressions. First, we can take the Boolean values, *true* and *false*, as *literal* expressions. Second, we can take *Boolean variables*,

such as *P* or *Q*, as a Boolean *variable* expressions. Finally, we take take each Boolean operator as having an associated expression constructor that takes one or more smaller *Boolean expressions* as arguments.

Notice that in this last step, we introduced the idea of constructing larger Boolean expressions out of smaller ones. We are thus defining the set of all Boolean expressions *inductively*. For example, if *P* is a Boolean variable expression, then we can construct a valid larger expression, $P \wedge true$ to express the conjunction of the value of *P* (whatever it might be(with the value, *true*. From here we could build the larger expression, *P lor (P land true)*, and so on, ad infinitum.

We define an infinite set of "variables" as terms of the form mkVar(s), where s, astring, represents the name of the variable. The term mkVar("P"), for example, is our way of writing "the var named P."

```
datatype Bvar = mkVar(name: string)
```

Here's the definition of the *syntax*:

```
datatype Bexp =
    litExp (b: bool) |
    varExp (v: Bvar) |
    notExp (e: Bexp) |
    andExp (e1: Bexp, e2: Bexp) |
    orExp (e1: Bexp, e2: Bexp)
```

Boolean expresions, as we've defined them here, are like propositions with paramaters. The parameters are the variables. Depending on how we assign them *true* and *false* values, the overall proposition might be rendered true or false.

9.9.2 Interpretation

Evaluate a Boolean expression in a given environment. The recursive structure of this algorithm reflects the inductive structure of the expressions we've defined.

```
type interp = map<Bvar, bool>
```

9.9.3 Semantics

```
function method Beval(e: Bexp, i: interp): (r: bool)
{
    match e
    {
        case litExp(b: bool) => b
        case varExp(v: Bvar) => lookup(v,i)
        case notExp(e1: Bexp) => !Beval(e1,i)
        case andExp(e1, e2) => Beval(e1,i) && Beval(e2, i)
        case orExp(e1, e2) =>  Beval(e1, i) || Beval(e2, i)
    }
}
```

}

Lookup value of given variable, v, in a given interpretation, i. If there is not value for v in i, then just return false. This is not a great design, in that a return of false could mean one of two things, and it's ambiguous: either the value of the variable really is false, or it's undefined. For now, though, it's good enough to illustate our main points.

```
function method lookup(v: Bvar, i: interp): bool
{
    if (v in i) then i[v]
    else false
}
```

Now that we know the basic values and operations of Boolean algebra, we can be precise about the forms of and valid ways of transforming *Boolean expressions.* For example, we've seen that we can transform the expression *true and true* into *true*. But what about *true and ((false xor true) or (not (false implies true)))*?

To make sense of such expressions, we need to define what it means for one to be well formed, and how to evaluate any such well formed expressions by transforming it repeatedly into simpler forms but in ways that preserve its meaning until we reach a single Boolean value.

9.9.4 Models

9.10 Satisfiability, Validity

We can now characterize the most important *open question* (unsolved mathematical problem) in computer science. Is there an *efficient* algorithm for determining whether any given Boolean formula is satisfiable?

whether there is a combination of Boolean variable values that makes any given Boolean expression true is the most important unsolved problem in computer science. We currently do not know of a solution that with runtime complexity that is better than exponential the number of variables in an expression. It's easy to determine whether an assignment of values to variables does the trick: just evaluate the expression with those values for the variables. But *finding* such a combination today requires, for the hardest of these problems, trying all :math:`2^n` combinations of Boolean values for *n* variables.

At the same time, we do not know that there is *not* a more efficient algorithm. Many experts would bet that there isn't one, but until we know for sure, there is a tantalizing possibility that someone someday will find an *efficient decision procedure* for Boolean satisfiability.

To close this exploration of computational complexity theory, we'll just note that we solved an instances of another related problem: not only to determine whether there is at least one (whether *there exists*) at least one combination of variable values that makes the expression true, but further determining how many different ways there are to do it.

Researchers and advanced practitioners of logic and computation sometimes use the word *model* to refer to a combination of variable values that makes an expression true. The problem of finding a Boolean expression that *satisfies* a Boolean formula is thus somtetimes called the *model finding* problem. By contrast, the problem of determining how many ways there are to satisfy a Boolean expression is called the *model counting* problem.

Solutions to these problems have a vast array of practical uses. As one still example, many logic puzzles can be represented as Boolean expressions, and a model finder can be used to determine whether there are any "solutions", if so, what one solution is.

9.11 Logical Consequence

Finally, logic consequence. A set of logical propositions, premises, is said to entail another, a conclusion, if in every interpretation where all of the premises are true the conclusion is also true. See the file, consequence.dfy, for a consequence checker that works by exhaustive checking of all interpretations. <More to come>.

CHAPTER

TEN

10. NATURAL DEDUCTION

Note to self: The next few chapters separate complexities on the way to full first-order logic. The first, addressed here, is the shift from a semantic to a syntactic approach to judging truth. Derivation vs. Evaluation.

We will use the reasoning principles just validated semantically to formulate analogous syntactic rules: i.e., natural deduction. These rules provide a needed alternative to truth tables for ascertaining truth in propositional logic. Truth tables grow too large too fast.

The next two chapters introduce, respectively, predicate logic without quantifiers but including interpretations over arbitrary sets; and then the introduction of quantifiers. [FIX BELOW: UNDER CONSTRUCTION.]

One way to define a set of *inference* rules that define ways that one can transform one set of expressions (premises) into another (a conclusion) in such a manner that whenver all the premises are true, the conclusion will be, too.

Why would anyone care about rules for transforming expressions in abstract languages? Well, it turns out that *syntactic* reasoning is pretty useful. The idea is that we represent a real-world phenomenon symbolically, in such a language, so the abstract sentence means something in the real world.

Now comes the key idea: if we imbue mathematical expressions with real-world meanings and then transform these expression in accordance with valid rules for acceptable transformations of such expressions, then the resulting expressions will also be meaningful.

A logic, then, is basically a formal language, one that defines a set of well formed expressions, and that provides a set of *inference* rules for taking a set of expressions as premises and deriving another one as a consequence. Mathematical logic allows us to replace human mental reasoning with the mechanical *transformation of symbolic expressions*.

CHAPTER

ELEVEN

11. SETS AS DOMAINS: PREDICATE LOGIC

In this chapter we move from propositional to a predicate logic, in which the values of variables can range not only over :math:{matbb B}‘ but over arbitrary sets. The issue of *validity* is complicated as it now has to be understood as involving judgements of truth that are independent of any particular interpretation.

CHAPTER

TWELVE

12. QUANTIFICATION: FIRST-ORDER LOGIC

We now address quantifiers with some care. We've been seeing and using them all along, of course. What we do in this chapter is to address them more rigirously. We discuss elimination and induction rules, and the fundamental concepts of induction principles and their use to build proofs by induction We'll then address quantification in practice: e.g., in Dafny and maybe SQL. Quantifiers also complicate automated verification systems because they allow for the expression of very complex specifications. We will briefly address ways in which programmers can think about helping Dafny when it needs guidance about where to look, e.g., for witnesses needed to prove existentials.

12.1 Universal and Existential Quantification

12.2 Introduction and Elimination Rules for Exists

12.3 Introduction and Elimination Rules for Forall

12.4 Induction Principles and Algebraic Data Types

This takes us in particular to induction principles and proofs (where we redeem our earlier aside on the injectivity of constructors). This is how we introduce truth claims with *for alls* elements in a given domain (or type), particularly in cases where the domain quanitified over is infinite.

As an example, we'd like to be able to deduce (prove) that every program written in some new language either type checks and is accepted by the type checker, and in this case no runtime errors can occur, or that it is expressly rejected by the checker, and that the checker will never *get stuck*.

How might we every prove something about all programs in a language?

CHAPTER

THIRTEEN

INDICES AND TABLES

- genindex
- modindex
- search