Mathematical Logic in Software Development Documentation

Release 1

Kevin Sullivan

Feb 13, 2018

CONTENTS:

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

To get a clear sense of the potential differences in performance between a pure functional program and an imperative program that compute the same function, 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* 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 Fibonacci number on each iteration, until we've got the result.

Footnote: by convention we index sequences starating 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 in a functional language, we can even run it if we wish. However, it might not give us the performance we require. An imperative program, by constrast, is *code*. It's almost like it's encrypted. It's certainly often cryptic. But it can be very efficient when run.

To get a sense of performance diferences, 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.

Start with 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*, to obtain the next one 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? It's easy to see that 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 it 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.

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 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: $\{(0, 1), (1, 1), (2, 3), ...\}$.

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 costing *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 one 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: $\{(0,1),(1,1),(2,3),(3,5),(4,9),(5,15),...\}$.

It reasonably easy to see that 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*.

Now that we see the formula, we can quickly compute the following values of cost as a function 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*;l and for *fib(11), 286* function evaluations. The cost measured by the number of function evaluations is growing much more quickly than n itself.

With our imperative program, the number of loop body interations grows *linearly* in *n*. We could say that the computational cost of running the imperative program to compute *fib(n)*, let's call it *cost(n), is *n+1*. How does the cost of the pure functional program compare? One thing to notice is that the cost of computing a Fibonacci element using our iterative approach 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, and thus also the cost of computing it recursively, grows at an *exponential* rate in *n*, with an exponent of about *1.6*. Increasing *n* by *1* doesn't quite double the cost of computing *fib(n)*, but it does multiply it by a factor of about *1.6*.

No matter how small the exponent, exponential functions eventually grow very large very quickly. In fact, in the limit, any exponential function (with any exponent greater than one) grows faster than any polynomial function no matter how high in rank and no matter how large the coefficients. The exponential-in-*n* cost of our beautiful but inefficient functional program grows vastly faster than the linear cost of our ugly but efficient imperative program as we increase *n*. For any reasonably large value of *n* (like 1000), it will be highly impractical to use the pure functional program, computation, whereas the imperative program will run quickly for values of n in the billions.

You can already see that the cost to compute *fib(n)* recursively for values of *n* larger than just ten or so is vastly greater than the cost to compute it iteratively. The math (the recursive definition clear but inefficient. The program is efficient, but woefully not transparent as to its function. We need the latter program for practical computation. But how do we ensure that hard to understand imperative code flawlessly implements the same function that we expressed so elegantly in mathematical logic and its computational expression in pure functional programming?

We address such problems by combining a few ideas. First, we use logic to express *declarative* specifications that precisely define *what* a given imperative program must do, an in particular what results it must return as a function of the arguements it received.

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

Second, we implement the specified program in an imperative language in a way that supports logical reasoning about its behavior. What kind of support is needed to facilitate logical reasoning is broached in this chapter. For example, we have to specify not only the desired relationship between argument and result values, but also how loops are designed to work in our code; and we 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.

We develop these idea in this chapter. 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 out program. Doing this requires that we deepen the way we understand loops. We end with a detailed presentation of the design and verification of an imperative program to compute elements 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 careful design of a loop is once again the very heart of

the problem. We will see how Dafny can help us to reason rigorously about loops, and that, with just a bit of help, it can reason about them for us.

5.1 Logical Specification

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)*.

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

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*).

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

A binary relation in ordinary mathematics is just a set of pairs of values. A function is a special binary relation with at most one pair with a given first value. A function is said to be a *single-valued* relation.

For example, pairs of non-negative integers in the relation that constitutes the factorial function include $(0, 1), (1, 1), (2, 2), (3, 6), (4, 24)$ and $(5, 120)$, but not $(5, 25)$.

On the other hand, square root is a relation but not a *function*. It is not singled valued. Both $(4, 2)$ and $(4, -2)$, two pairs with the same first element but different second elements, are in the relation. That is because both *2* and *-2* are squarer roots of *4*. The *positive square root* relation, on the other hand, is a function, comprising those pairs in the square root relation where both elements are non-negative. It thus includes $(4, 2)$ but not $(4, -2)$.

We could formulate the square root *relation* as a *function* in a different way: by viewing it as a relation that associates with each non-negative integer the single *set* of its square roots. The pair $(4, \{2, -2\}$ is in this relation, for example. The relation is now also a function in that there is only one such pair with a given first element.

Now what we mean when we say that a program computes a function or a relation is that whenever it is given a valid argument representing the *first* value of a pair in the relation, it computes a *second* value such that the pair, $(first, second)$ is in the given relation. When we say, for example, that a program *computes the factorial function*, we mean that if we give it a non-negative number, *n*, it returns a number *m* such that the pair *(n,m)* is *in* the relation. And for *(n,m)* to be in the relation it must be that $m = fact(n)$. The program thus has to return $fact(n)$.

A program that computes a *function* is deterministic, in the sense that it can return at most one result: because there is at most one result. When a program computes a relation that is not a function, it can return any value, *m*, where *(n,m)* is in the specified relation.

5.2 Rigorous 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.

In addition to choosing a language with features that help to support formal reasoning (such as strong, static typing), we sometimes also aim to write imperative code in a way that makes it easier to reason about formally (using mathematical logic). As we will see below, for example, the way that we write our while loops can make it easier or harder to reason about their correctness.

5.3 Formal Verification

Our ultimate aim to deduce that, as written, a program satisfies its input-output 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 in this sense (that if it's started in a state that satisfies the assertion, *P*, and if it terminates then, it will do so in a state that satisfies *Q*) as PCQ. 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 specification.

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* amd *Q* and that then formally, and to a considerable extent automatically, verifies *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 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.

5.4 Typical Implementation of the Factorial Function

So far the material in this chapter has been pretty abstract. Now we'll see what it means in practice. To start, let's consider an ordinary imperative program, as you might have written in Python or Java, for computing the factorial function. The name of the function is the only indication of the intended behavior of this program. There is no documented 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;
}
```

Sadly, this program 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?

The problem is that the program lacks a complete 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 *right* even means.

Now let's see what happens when we make the specification complete. The precondition will continue to be expressed by the type of the argument, *n*, being *nat*. However, we have added a postcondition that requires the return result to be the factorial of *n*. Note that we used our functional definition of the *factorial* function in the *specification* of our imperative code. The pure functional program is really just a mathematical definition of factorial. What we assert with the postcondition is thus that the imperative program computes the factorial function as it is defined in pure mathematics.

```
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;
}
```

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

5.5 A Verified Implementation of the Factorial Function

Here's verified imperative program for computing factorial. We start by documenting the overall program specification.

```
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, as 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, and sequences of commands the program took to get here.

```
assert n > 0;
```

Strategy: use a while loop to compute the answer. 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 down, 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 *i*, and the value of *i* represents how much of this 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 the task of multiplying *a* by the factors from *8* down to *1* remains to be done. A critical "invariant" then is that if you multiply *a* by the factorial of *i* you get the final answer, the factorial of *n*. And in particular, when *i* gets down to *0*, *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.

Step 1. Set up state for the loop to work. We first initializie a := 1 and i := n and check that the invariant holds. Note that we are using our pure functional math-like definition of fact as a *specification* of the factorial function we're implementing.

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

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 in the execution of this program.

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

Step 2: Now evaluate the loop to get the answer. To evaluate a loop, first, evaluate the loop condition (i > 0).Then , if the result is false, terminate the loop. Otherwise, evaluate the loop body, then iterate (run the loop again, starting by evaluating the loop condition).

Note that we can deduce that the loop body is going to execute at least once. 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. But we can now do better than just reasoning in our heads; we can use logic to express what we believe to be true and let Dafny try to check it for us automatically.

```
assert i > 0;
```

Let's just think briefly about cases. We know i can't be zero. It could be one. If it's one, then the loop body will run. The loop body will run. a, which starts at 1, will be multiplied by i, which is 1, then i will be decremented. It will

have the value 0 and the loop will not run again, leaving a with the value 1, which is the right answer. So, okay, let's run the loop.

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

At this point, we know that the loop condition is false. In English, we'd say it is no longer true that i is greater than zero." We can do better that saying this in natural language then forgetting it. We can use formal logic to formalize and document our belief and if we do this then Dafny pays us well for our effort by checking that our assertion is true.

```
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. And that's what we'd expect, because that's the condition on which the loop terminates, which is just did! But better than just saying it, let us also formalize, document, and check it.

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

Now it's easy to see. No matter what value i has, a * fact(i) == fact(n), and i == 0, so we have a * fact(0) == fact(n), and we know that fact(0) is 1 because we see that in the very mathematical definition of fact, so it must be that a = fact(n). Dafny can check!

```
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.

5.6 A Verified Implementation of the Fibonacci Function

Similarly, here an imperative implementation of the fibonacci function, without a spec.

```
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)* and *fib1* will store the value of *fib(1)*.

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

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, which is to say that we know, 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 then we would already have returned if it were *0* or *1*, to it must now be *2* or greater. We can document the belief that the current state of the program has to property that the value of the variable *n* is greater than or equal to *2*, and Dafny will verify this assertion 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 to use a while loop with an 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 computed fib(i) and that its value is stored in fib1. We've already assigned the value of fib(0) to fib0, and of fib(1) to fib1, so to set up the desired state of affairs, we should initialize *i* to be *1*.

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

We can state and Dafny can verify a number of conditions that we expect and require to hold at this point. First, *fib1* equals *fib(i)*. Now to compute the next (*i+1*) 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, and indeed we do have that 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, as *i* is *1* and *n* is at least *2*, so the condition *i* < *n* is *true*, which dictates that the loop body must be evaluated.

Within the loop body we'll compute fib(i+1) (we call it *fib2* within the loop) 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*.

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 still *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'll also have that *i* == *n* (it takes a little reasoning to prove this), 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.

That's our strategy. So let's go. Here's the while loop that we have designed. And here, for the first time, we see something crucial. We tell Dafny about certain properties of the state of the program that hold both before and after every execution of the loop body. We call such properties *invariants*. Dafny needs to know these invariants to prove to itself (and to us) that the loop does what it is intended to do: that the result at the end will be as desired.

```
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 *within* the execution of the loop body, but they do have to hold before before and after each execution.

The body of the loop is just as we described it above, and we can use our own minds to deduce that if the invariants hold before the 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*. This is 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, and logic is right. 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 and we have recapitulated the proof in this sequence of assertions) that we have without a doubt computed the right answer. Dafny has also proved to itself that the loop always terminates, and so we have in effect 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;
```

5.7 What is Dafny, Again?

Dafny is a cutting-edge software language and tooset developed at Microsoft Research—one of the top computer science research labs in the world—that provides such a capability. We will explore Dafny and the ideas underlying it in the first part of this course, both to give a sense of the current state of the art in program verification and, most importantly, to explain 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; for generating proofs is exactly what tools like Dafny do *under the hood*.

A well educated computer scientist and a professionally trained software developer 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
```

Return (terminate execution of a method)

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.

We illustrate the use of return at the end of this program.

```
return;
```

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. TOWARD LOGIC: BOOLEAN ALGEBRA

As a stepping stone toward a deeper exploration of deductive logic, we explore the related notion of Boolean *algebra*. Boolean algebra is akin to ordinary high school algebra, and as such, deals with values, variables, and functions. However, the values in Boolean algebra are limited to the two values in the set, $bool = \{0, 1\}$. Based on the connection between Boolean algebra and *propositional logic*, these values are often written as *false* and *true*, respectively.

7.1 The *bool* Type 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. Here's a (useless) Dafny method that illustrates how Boolean values 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 values and 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.

7.2 Boolean Algebra

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.

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.

7.2.1 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.

7.2.2 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

7.2.3 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.

7.2.4 A Ternary Binary Operator

Finally, to emphasize the point that there are binary operations of all arities, we introduce just one ternary Boolean operator. It takes three Boolean 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 it works is that the value of the first argument determines which of the next two arguments values the function returns. If the first argument is *true* then it returns the second argument, otherwise it returns the third. So, for example, *ifThenElse_{bool}(true, true, false)* returns true.

It is sometimes helpful to break up the name of such a function into parts (if, then, and else(and to spread out the arguments between the parts. So *ifThenElse_{bool}(true, true, false)* could be written as *if true then true else false.* The link to conditional expressions and commands in everyday programming should clear.

As an exercise for the reader: How many ternary Boolean operations are there? Hint: an operator with *n* Boolean arguments has 2^n possible 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?

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

7.3 Logic: Formal Languages for Syntactic Reasoning

Mathematical logic is grounded in the definition of valid sentences in formal languages and the definition of rules for transforming one such sentence into another in a valid way. For example, *2 + 2* is a valid expression in the languages of arithmetic (but *2 2 + +* isn't), and the rules of mathematical logic allow this expression to be replaced with the expression, *4*, but not by *5*.

Why would anyone care about precisely defined transformations between 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*.

To define a logic, we only have to say how to form valid expressions, and what the rules are for transforming such expression in valid ways. To use such a logic for practical good, which is the ultimate aim, of course, we have to learn the art of figuring out how to represent the real-world phenomena of interest into symbolic forms in ways that will allow us then to use the available transformation rules repeatedly to deduce a new expression, one that holds the answer to the question we asked, or the result we need to act upon to have some desired effect in and on the world.

7.4 Boolean Algebra, Expressions, and Decision Problems

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.

7.4.1 The Syntax of Boolean Expressions

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.

7.4.2 Interpretations in Boolean Logic

7.4.3 The Semantics of Boolean Expressions

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>
```

```
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.

7.4.4 Interpretations

7.4.5 Validity, Satisfiability, Unsatisfiability

7.4.6 The Most Important Unsolved Problem in Computer Science

It is now possible for you to understand what is 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.

CHAPTER

EIGHT

SETTING UP TO USE GIT

8.1 Git and Distributed Software Development

Large systems built by hundreds or thousands of developers around the world

- Challenge is to coordinate many changes being made around the globe
- Often have a shared, reference copy that everyone shares (e.g., on GitHub)
- Can't really have thousand of people making uncontrolled changes
- Rather, everyone has entire history of whole project on local machine
- Anyone can edit their own copy
- Anyone with privileges can push local changes back to central version
- Anyone can ask central developers to "pull" local changes into central repo
- Anyone can register an "issue" with main developers
- Updates made to central repo can then be "pulled" down to local versions
- But git won't overwrite local changes – it will say "there's a conflict"

8.2 Setting Up

Let's make sure everyone is set up with git and up-to-date copy of cs-dm repo

- Open a terminal window [check; if not, install MSYS2 per instructions]
- Run "git" command – does it run? [check; if not, install git]
- Change into directory you're using for this class [advice, no spaces; check?]
- List files in this directory; ls.
- Rename any current cs-dm directory to cs-dm.old
- Clone current cs-dm directory
- Rename to cs-dm-reference; you will NOT make local changes here
- Clone current cs-dm directory again; rename to cs-dm-hw3
- Open cs-dm-hw3/dafny folder in VS Code; you're set to do next HW

8.3 A Demo

Let's see how git works; I will fix errors in code; I will push; you will pull; that's it!

- Point browser at cs-dm-reference/notes/_build/html/index.html
- Open separate browser window on GitHub repo
- Let's look at the issues; and fix them!
- In notes, let's find mistake
- Now I will fix it, rebuild the HTML and PDF files, and push an update to GitHub
- Now you should cd into cs-dm-reference directory (the one you won't edit)
- Do a git pull to pull the changes now on GitHub into your local repo
- And refresh browser to display the now updated HTML files!
- Note: You should do a git pull in each copy of the repo needing an update

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search