Mathematical Logic in Software Development Documentation

Release 1

Kevin Sullivan

Jan 29, 2018

CONTENTS:

CHAPTER

ONE

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, on 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

LOGIC AND CODE

We've discussed requirements, specifications, and implementations as software artifacts serving distinct purposes. For good reasons, these artifacts are generally written in different languages. In this unit, we discuss these different kinds of languages—mathematical logic for specifications and imperative languages for code—why they are used for different purposes, the fundamental advantages and disadvantages of each, and why modern software development requires fluence in and tools for handling artifacts written in multiple such languages.

2.1 Imperative Implementations and Declarative Specifications

The language of implementations is code in what we call an *imperative programming language*. Examples of such languages include Python, Java, C++, and Javascript. The most salient property of such a 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 such languages operate by (1) reading, computing with, and updating values stored in a *mutable memory*, and (2) interacting with the world outside of the computer by executing input and output (IO) commands.

The language of formal requirements and specifications, on the other hand, is some kind of *mathematical logic*. Examples of logics that we will study and use include *propositional* and *predicate* logic. An example of a kind of logic important in software development but that we will not study in this class is *temporal logic.*

For purposes of software specification, the most salient property of such a logical language is that it is *declarative*. Expressions in logic will state *what* properties or relationships must hold in a given situation, particularly how results must relate to inputs, without providing executable, step-by-step procedures describing *how* to actually compute such relationships.

To make the difference between procedural and declarative styles of description clear, consider the problem of computing the positive square root of a given non-negative number, *x*. We can *specify* the answer in a clear and precise logical style by simply stating that, for any given non-negative number *x*, we require a value, *y*, such that $y^2 = x$. We would write this mathematically as $\forall x \mid x >= 0, sqrt(x) = y | y^2 = x$. In English, we'd pronounce this formula as, "for any *x* where *x* is greater than or equal to zero, the square root of *x* is a value *y* such that *y* squared is equal to *x*."

We now have a *declarative specification* of the desired relationship between *x* and *y*. What we don't have, however, is a step-by-step *procedure* for computing this relation by finding a value of *y* for any given value of *x*. You can't just run a specification written in the language of mathematical logic.

The solution is to shift from mathematics as a specification language to imperative code as an implementation language. In such a language, we then craft a step-by-step procedure that, when run, computes the results we seek. Here's an example of a program in the imperative language, Python, for computing positive square roots of non-negative numbers using Newton's method.

```
def sqrt(x):
    """for x>=0, return non-negative y such that y^2 = x"""
    estimate = x/2
```

```
    while True:
        newestimate = ((estimate+(x/estimate))/2)
        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 deep problems with this approach. First, as we've discussed, natural language is subject to ambiguity, inconsistency, 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 (changes over time), and in their rush to ship code, developers often forget, or neglect, to update comments. So, in practice, even if a given procedure is initially consistent with a specification given as comment, inconsistencies can and often do develop over time.

2.2 Why Not a Single Language for both 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 very 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* (at least not efficiently).

On the other hand, imperative code states *how* a computation is to be carried out, but enerally doesn't make clear *what* it's computing. You would be hard-pressed, based on a quick look at the Python code above, to explain *what* it does (but for the fact that we embedded the spec into the code as a doc string).

We are driven to a situation in which we have to express what we want and how to get it, respectively, in very different languages. This situation creates a difficult new problem: to verify that a program written in an imperative language satisfies a specification written in a declarative language. This is the problem of *verification*. Have we built a program right (where right is defined by a specification)?

CHAPTER

THREE

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 simpl 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 peice of code that assumes that y doesn't change unless a change is made expliticly 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

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, $\forall x \in \mathbb{Z}, x + 1$. 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.

4.4 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!

4.5 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 is 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

INTEGRATING FORMAL SPECIFICATION WITH IMPERATIVE PROGRAMMING

An important approach to solving such problems is to enable the integration of *formal specifications* with imperative programming code along with mechansims (based on *logical proof* technology) for checking the consistency of code with specifications. Specifications are given not as comments but as expressions in the language of logic right along with the code, and checkers attempt to verify that code satisfies its corresponding *specs*.

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.

5.1 To integrate

5.2 Fitting it All Together

So as we go forward, here's what we'll see. Ultimately, for purposes of efficiency and interactivity (I/O), we will write imperative code to implement software systems. That said, we can often use functional code to implement subroutines that perform computations that do not require mutable storage or I/O. We will *also* use pure functional programs as parts of *specifications*.

For example, we might specify that an *imperative* implementation of the factorial function must take any natural number *n* as an argument and return the value of *fact(n),* our *functional* program for the factorial function. The logical specification of the imperative program will be an *implication* stating that if a proper argument is presented, a correct result *as defined by a functional program* will be produced.

We can thus use pure functional programs both for computation *when appropriate*, yielding certain benefits in terms of understandability and safety, and as elements in logical specifications of imperative code. In Dafny, a pure functional program that is intended only for use in specifications is declared as a *function*. A pure functional program intended to be called from imperative code is declared as a *function method.* Imperative programs are simply declared as methods.

Here's a complete example: an imperative program for computing the factorial function with a specification that first requires *n>0* and that then requires that the result be *fact(n)* as defined by our functional program.

```
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 * t;
        t := t - 1;
    }
    f := a;
}
```

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

Unfortunately Dafny reports that it cannot guarantee—formally prove to itself—that the *postcondition* (that the result be right) will necessarily hold. Generating proofs is hard, not only for people but also for machines. In general it's impossibly hard, so the best that a machine can do in practice is to try its best. If Dafny fails, as it does in this case, what comes next is that the developer has to give it some help. This is done by adding some additional logic to the code to help Dafny see its way to proving that the code satisfies the spec.

We'll see some of what's involved as we go forward in this class. We will also eventually dive in to understand what proofs even are, and why in general they are hard to construct. Lucky for mathematicians! If this weren't true, they'd all be out of jobs. Before we go there, though, let's have some fun and learn how to write imperative code in Dafny.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search