Mathematical Logic in Software Development Documentation

Release 1

Kevin Sullivan

Jan 27, 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 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-ede 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*.

2.3 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)?

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.

2.4 Pure Functional Programming as Runnable Mathematics

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.

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++.

To see how functional languages allow one to implement functions in ways that closely mirror their mathematical definitions, consider the factorial function and an implementation of this function in the functional *sub-language* of Dafny. (Dafny provides sub-languages for specification and for both functional and imperative programming.)

The factorial function is defined recursively. For any natural (non-negative whole) number, *n, factorial(n)* is defined by two cases: one for when *n* is zero, and one for any other value of *n*.

$$f(x) = \begin{cases} 1, & \text{if } x < 0. \\ 0, & \text{otherwise.} \end{cases}$$

First, if *n = 0* (called the *base case*) then *factorial(n)* is defined to be 1. Otherwise, for any *n* where *n > 0)*, *factorial(n)* is defined recursively as *n * factorial(n-1)*. This is what we call the *recursive* case. By recursive, we mean that the function is used in its own definition.

Recursive definitions are ubiquitous in mathematics. In fact, if you get right down to it, most every function you've ever thought about is defined recursively. For example, the addition of two natural (non-negative) numbers *m* and *n* is defined recursively. If *m = 0*, the base case, then the answer is *n*. If (m>0), the recursive case, then there is some natural number *m'*, the *predecessor* of *m*, and in this case the result is one more than (the successor of) the sum of *m'* and *n*. such that *m = m'+1*. Recursion is thus fundamentally a mathematical and not (just) a computational concept.

The reason that such definitions makes sense, and are not just endless self loops, is that they are *well-founded*. What this means is that for any given *n* (a natural number), no matter how large, the looping eventually ends. For example, *fact(3)* is defined to be *3 * fact(2).* Expanding the definition of the recursive call to the *fact This is *3 * (2 * fact(1)). This in turn is *3 * 2 * 1 * fact(0).* Because *fact(0)* is a base case, defined to be just *1* without any further recursion, the recursion terminates, and the end result is *3 * 2 * 1 * 1*, which finally evalutes to *6*. o matter how large *n* is, eventually (in a finite number of steps), the recursion will bottom out at the base case, and a result will be produced.

Our functional program to compute the factorial function mirrors the abstract mathematical definition. The program, like the definition, is recursive: it *uses* (is defined in terms of) itself. Here's the code in Dafny's functional programming sub-language.

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

The program takes a value, *n*, of type int (any integer). Then the requires *predicate* (a piece of logical specification) restricts the value of *n* to be non-negative. Finally you have the recursive rules for computing the value of the function. If *n* is zero the result is one otherwise it's *n* times the function itself applied to *n-1*.

So here you have something very interesting. First, the code is just like the mathematics. Functional programming languages are not nearly as expressive as predicate logic (as we'll see when we really get to logic and proofs), but they

are much closer to mathematics, in many cases, than imperative code. Programs in pure functional languages are more expressive and easier (for humans and machines) to reason about than programs written in imperative languages.

Second, we now see the integration of logic and code, The *requires* predicate is a logical proposition *about* the value of the parameter, *n*, expressed not as a comment but as a formal and machine-checkable part of the program.

Althird, though you can't see it here in this document, Dafny checks to ensure that no code ever calls this function with a value of *n* that is less than zero, *and* it proves to itself that the recursion is well founded. That is a lot more than you could ever expect to get programming in an imperative language like Python.

Pure functional programming languages thus provide a way to program functions almost as if in pure mathematics. At the same time, such programs can be run reasonably efficiently and analyzed by human and machanized checkers.

So what's the downside? Why not always program in such languages? One reason is efficiency. It's a challenge to get programs in such languages to execute efficiency precisely because there is no notion of a mutable memory. There's thus not way (conceptually) to update a part of a large data structure; rather one must write a function that takes a given data structure and that computes and builds a whole new one, even if it differs from a given data structure only a little.

A second, even more fundamental limitation, is that there is no concept of interacting with an external environment in the realm of pure functions. You've got data values and functions that transform given values into new values, and that's it. You simply cannot do I/O in a pure functional language! There are functional languages that are meant for practical programming (such as Haskell), in which you can of course do I/O, but the capabilities to do I/O are non-functional. They are in a sense *bolted on*. They are bolted on in clever, clean ways, but the fact remains that I/O is just not a functional concept.

2.5 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: int) returns (f: int)
  requires n>= 0
  ensures f == fact(n)
{
  if (n==0)
  {
      f:= 1;
      return;
  }
  var t := n;
  var a := 1;
```

```
  while (t !=  0)
  {
      a := a * t;
      t := t - 1;
  }
  f := 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

THREE

INDICES AND TABLES

- genindex
- modindex
- search