# Modularity in mathematics*

Jeremy Avigad

September 23, 2017

**Abstract**

In a wide range of fields, the word "modular" is used to describe complex systems that can be decomposed into smaller systems with limited interactions between them. This essay argues that mathematical knowledge can fruitfully be understood as having a modular structure, and explores the ways in which modularity in mathematics is epistemically advantageous.

## Contents



*This work has been partially supported by Air Force Office of Scientific Research MURI FA9550-15-1-0053. Versions of this material were presented to a philosophy of mathematics seminar run by Kenneth Manders at the University of Pittsburgh in 2014, and to the *Current Issues in the Philosophy of Practice of Mathematics and Informatics* workshop in Toulouse in 2016. I am grateful to both audiences for helpful discussions. I am especially grateful to Yacin Hamami, Robert Lewis, Rebecca Morris, and David Waszek for numerous comments, suggestions, and corrections, and a good deal of moral support.

# 1 Introduction

Roughly speaking, a complex system is said to be *modular* when it can be decomposed into smaller systems, or *components*, with limited or controlled interactions between them. The term "modular" is now used in a number of fields, including biology [14], computer hardware and software design [9, 19, 31, 32, 41], business administration [37], and architecture, as well as research at the intersection of neurobiology, cognitive science, psychology, and philosophy of mind [22, 10, 15, 34].

The thesis of this essay is that mathematical knowledge is structured in modular ways. Part of the project is descriptive, in the sense that I will offer a certain perspective on the constituents of mathematical knowledge and clarify the ways in which they can be said to be modular. But there is also a normative component, in that I also aim to explain why this *should* be the case, by highlighting some of the epistemological benefits that such a modular structuring confers.

The concept of modularity is often applied to the study of natural systems, sometimes with an eye toward proving an explanation as to why these systems have evolved the way they have. But the concept is equally often used in discussions of manufactured systems, where it is often portrayed as an explicit design goal. Modularity is generally said to improve the comprehensibility of such systems and lead to greater robustness, flexibility, efficiency, and economy. Here we will explore the extent to which the design of mathematical resources can be understood in such terms.

In a branch of computer science known as *formal verification*, one can now use computational proof assistants to verify the correctness of substantial mathematical theorems [6]. This affords a two-step process for translating the concept of modularity from software engineering to mathematics. Insofar as the formal proof texts that serve as input to computational proof assistants are like computer programs, it makes as much sense to talk about modularity in formal mathematics as it does to talk about modularity in software design. And insofar as these formal proof texts illustrate important features of informal mathematical texts, in makes sense to talk about modularity in informal mathematics as well.

This strategy does not presuppose strong assumptions about the relationship between formal and informal reasoning. It can be seen, rather, as a heuristic means of making sense of the phenomena. Even if there were no formal languages or computational proof assistants, mathematical knowledge would still be modularly structured; but the availability of formal proof languages and their similarities to programming languages provide us with ready-made conceptual tools to begin to understand how and why this is so.

Section 2 lays the groundwork by introducing a general framework for thinking about mathematics and its goals. Section 3 analyses the way that modularity is understood in various sciences, with a focus on computer science, which is closest to our present concerns. Section 4 transports notions of modularity from computer science to mathematics. Section 5 considers examples that illustrate some of the ways that modularity plays out in everyday mathematics. Finally, Section 6 suggests directions for future research, and sums up the main conclusions.

# 2 Mathematics from a design standpoint

## 2.1 Mathematical resources

Let us start with the question as to what mathematics is, and, more to the point, what sorts of objects can bear the predicate "modular." In the approach we will adopt here, we will think of mathematics as a shared linguistic practice, so that the objects of evaluation are things like definitions, theorems, proofs, problems, conjectures, questions, theories, and so on. These are all things that can be written down, and, indeed, things that are written down, constituting the mathematical literature. In modern logic, such objects are viewed as pieces of syntax. The way we will use the notions here will perhaps admit some degree of abstraction, factoring out the idiosyncrasies of a particular language or choice of expression. But if we are not dealing with raw syntax, we are dealing with something pretty close to it. Below I will try to spell out the sense in which a proof or a theory can be said to be modular, as well as the sense in which definitions and lemmas support modularity.

I have argued elsewhere, however, that limiting attention to such syntactic entities is too restrictive [3, 4, 5]. In order to address important epistemological issues, we need to make sense of more dynamic components of our understanding: things like methods, concepts, heuristics, and intuitions, which give rise to the abilities, or capacities, that we take to be constitutive of mathematical thought. The problem is that we do not yet have good ways of talking about these things, and so, for the most part, I will focus on the syntactic entities enumerated in the last paragraph. I expect that when we do have better means of thinking of mathematical knowledge in the broader sense, we will find that methods and concepts have a modular structure that is supported by the modularity of the syntactic entities. In any case, starting with the syntactic entities cannot hurt.

Any normative evaluation of mathematical resources necessarily presupposes some understanding of what it is we want them to do. It is generally held that mathematics is a means of getting us to the truth. This intuition that can be cashed out in semantic terms, which is to say, doing mathematics means trying to discover statements that are true in virtue of standing in an appropriate relation to the mathematical objects they refer to; or in epistemic terms, which is to say, doing mathematics means justifying one's claims in mathematically appropriate ways. Either viewpoint is consistent with the project set forth here, and we will not be directly concerned with the foundational task of coming to terms with the nature of mathematical justification and truth.

What is more important to the present work is the fact that simply getting to the truth cannot be the whole story. Otherwise, glib epistemic advice such as "check every natural number to ascertain the truth of the Goldbach conjecture" or "appeal to an omniscient and beneficent deity for the answer" would solve most of our philosophical problems. Implicit in all philosophical approaches to thinking about mathematics is the recognition that we are finite beings with finite resources, facts that constrain the epistemological account. The only additional observation we need is that not all finite burdens are equal: some burdens are bigger than others, which is to say, some tasks require greater epistemic resources.

As a result, we should think about mathematics as an attempt to get at the truth in an *efficient* manner. We want our definitions and theories to be simple so that we can understand them and deploy them more easily, and we want our proofs and solutions to be reasonably short, so that we can devote our energy to even harder problems and more difficult proofs. If a computation carried out on our fastest computer will not terminate before the sun burns out, the possibility of carrying out that computation is closed to us. A computation that gets us the right answer in reasonable time is therefore to be valued over one that does not.

Developing such a view requires one to come to terms with ways of measuring simplicity and complexity (see the discussion of this in [5]). This essay makes a small start on doing so, but without developing precise formal measures. For the moment, naïve intuitions on the nature of simplicity will have to suffice. It seems uncontroversial to say that it is usually easier to understand a short proof than a long one, at least given the right background knowledge; that it is easier to work through a proof that requires us to keep fewer pieces of information in mind at any given stage; and that it is easier to solve a problem when the context suggests which steps will plausibly lead to a solution, rather than trying all paths blindly. My goal here is to begin to explain how modularity can deliver such benefits.

To summarize: here we will view mathematics as a body of resources, both syntactic entities like definitions, theorems, proofs, and theories, as well as less-easily-circumscribed resources, such as concepts, methods, and heuristics. These resources are designed to help us get to the truth (answer questions, make predictions, solve problems, and prove theorems) simply and efficiently. Such resources are valuable insofar as they serve that purpose well, and the goal here

is to begin to understand some of the general principles that ensure that they do.

## 2.2 Mathematics and design

The use of the word "designed" in the last paragraph imposes a distinct way of thinking about mathematical resources, namely, as artifacts that are developed with particular ends in mind. Indeed, Kenneth Manders [30, 29] has used the term "artifact" in such a way, and his choice of terminology is apt. Whenever we introduce a new definition or notation, lay out a sequence of lemmas that make it possible to prove a theorem, or introduce a new algorithm or method of calculation, we contribute a new resource to the body of mathematical knowledge. Whenever we reformulate a definition, generalize a lemma, or rewrite a proof, we are, in effect, tinkering with those resources to augment their utility.

The exploration here is intended as a contribution to a theory of mathematical design — that is, a design theory for mathematics, one that can help us understand the principles that govern the effectiveness of a body of mathematical resources. Such theories are familiar across the arts and sciences: we have theories of automotive engineering, theories of software design, theories of architecture, theories of graphic design and typesetting. They help us understand what makes a good car, a good house, a good program, or a good poster, and articulate guidelines that help ensure that these artifacts serve their purposes well. A theory of mathematical design should do the same for mathematics.

Some comments may forestall misunderstanding. To start with, developing a theory of mathematical design is not sharply distinct from doing mathematics. There are design decisions implicit in every mathematical offering. Developing the philosophy of mathematics as a design science is a matter of reflecting on the mathematical choices we make, and then "going scientific": articulating the goals with greater precision, modeling the space of design options, and assessing their effects. The analogy to automotive engineering may be helpful: while humans have designed vehicles for centuries, the theory of automotive engineering aims to articulate the goals and constraints, understand the tradeoffs, and offer general methodological guidelines that contribute to the success of the design process.

If nothing else, such a design theory for mathematics may help us better convey our expertise, since the aim of our expository and educational efforts in mathematics is to convey advice to others that will help them do mathematics well. But the need is not only pedagogical. Just as twentieth century work in foundations has supported important developments in mathematical method, and, more recently, has supported the mechanization of mathematical reasoning and verification, so, too, can a theory of mathematical design contribute to mathematics itself. The primary goal here, however, is philosophical: mathematics is important to us, and we would like to understand how it works.

A concern commonly raised when it comes to the normative assessment of mathematical resources is that the bases for judgment may be contextual, depending, for example, on the capacities and goals of the agents that employ

them. Put simply, what counts as a useful piece of mathematics to you may be less valuable to me. Our evaluations may depend on our backgrounds: perhaps you know differential geometry, and I don't. It may depend on our talents and taste: perhaps you prefer geometric arguments, whereas I am better when it comes to algebraic manipulations. And it may depend on our individual goals: perhaps you are interested in the Riemann hypothesis, while I am trying to prove the Goldbach conjecture. This may leave us in the uncomfortable situation of trying to develop an objective science of something that is largely subjective, or at least highly dependent on context.

Once more, the analogy to the design sciences like automotive engineering is handy. What makes a good car depends strongly on the desires, attributes, and goals of the owner, and will vary depending on intended use: commuting to work, transporting a family, winning NASCAR races, or impressing a potential mate. But theories of automotive engineering tend to bracket these issues, relying on more objective measures of value: capacity, legroom, storage space, fuel efficiency, horsepower, acceleration, and the like. It is understood that there are tradeoffs involved, but the hope is that various weightings and combinations of these parameters are sufficiently capable of representing the more subjective measures to provide useful guidance as to how the latter can be addressed.

In the same way, we would expect a theory of mathematical design to be parameterized by features of the mathematical context that play a role in normative evaluations. Part of the challenge in developing such a theory is determining what these features are, and how they interact.

# 3 The concept of modularity

## 3.1 The general notion

The *locus classicus* of the study of modularity in complex systems is Herbert Simon's 1962 essay, "The architecture of complexity" [36], which examines the nature of complex systems in biology, physics, and economics, as well as social and symbolic systems. Though widely viewed as a seminal source in the study of modular systems, it is a curious historical fact that the word "modular" never appears in that work; rather, Simon used the phrase "nearly decomposible" in its place. The term "modular" is now used in disparate fields of research, and applied to both natural systems and systems that are designed. Before focusing on the use of the term in computer science, it will be helpful to try to discern features that are common to the various descriptions of modularity in the disciplines just enumerated.

It is important to keep in mind that any discussion of a complex system, whether it is natural or artificial, presupposes a level of description that is appropriate to the features and behaviors of interest. For example, a human being can be construed as a biological system, a cognitive system, or an agent in a social network. A different design description will be operant in each case. There is an inherently teleological component to the choice of a description; when we

speak of the design of a system, we invariably have a functional description of the system and its components in mind, and, moreover, are generally interested in understanding how the behavior of the components contribute to the system's observed or desired behavior. It will therefore be important, when we turn our attention to such mathematical artifacts later on, to think about the features and behaviors that we are trying to model. In the meanwhile, keep in mind that when we talk about a modular system, we are really talking about the modularity of a certain design description, which we take to be adequate to capture those properties that are of interest.

With this *caveat* implicit, a complex system is generally said to be modular to the extent it has the following features:

- The system is divided into *components*, or *modules*, with *dependencies* between them.
- The division supports a level of *abstraction*: the function of the components can be described vis-à-vis the functioning of the entire system, without reference to the particular *implementation*.
- Dependencies between modules are kept small, and mediated by precise *specifications*, or *interfaces*.
- Dependencies within a module may be complex, but, due to *encapsulation* or *information hiding*, these are not visible outside the module.[1]

The relevant notion of "dependency," which is central to this description, will depend on the kind of system under analysis. In an administrative system, dependencies can include channels of communication between and within components, as well as relationships of authority. In the design of systems hardware, the relevant dependencies are physical connections or data transfers between and within components; but they can also be used to model dependencies between activities and events involved in the factory production of the system. In a biological system, the relevant dependencies are likely to include causal relations between processes and subsystems. Below, we will discuss, in detail, the kinds of dependencies that are relevant to software design and to mathematics.

Modularity is often associated with an additional property:

- Organization into modules can be *hierarchical*: within a module, components can be divided into smaller *submodules*, and so on.

This is not a necessary feature of a modular system, in that one can have modular designs that are essentially flat.[2] But a hierarchical design only makes sense in terms of a modular presentation, and, conversely, the most modular description of a system is often obtained via a hierarchical conception of its components.

[1] Some characterizations of modularity are more involved. For example, Fodor [22] provides a long list of features generally associated with modularity. However, others have pointed out [10, 34] that most of these seem to be derivative of the notion of encapsulation.

[2] Parnas [32] observes this as well.

As mathematical theories and proofs also have a hierarchical structure, this is an issue that is worth keeping in mind.

With respect to both natural and artificial (which is to say, designed) systems, modularity is often credited with the system's ability to achieve a desired behavior. In the case of designed systems, modularity is also credited with making it possible for an agent to produce the system itself (thereby also, indirectly, achieving the desired behavior). Since we are thinking of mathematics as a human artifact, the analogies to designed systems will generally be more appropriate. Across the literature, the purported benefits of modular design generally fall under the following headings:

- *Comprehensibility.* When a system is modular, it is easier to understand, explain, and predict its behavior. In fact, modularity is often held to be a precondition for comprehensibility, or surveyability: when a system is sufficiently complex, it *cannot* be adequately understood unless a sufficiently modular description is available.

- *Reliability and robustness.* An appropriately modular description makes it possible to assess and test components of a system individually; to localize a problem to the behavior of one component; and to detect problems that would otherwise be lost in an overabundance of detail.

- *Independence.* A modular design allows the components of a system to be built (or to evolve) independently. They can be built concurrently, by different agents, at different locations.

- *Flexibility.* A modular design allows the system to change more quickly. For example, one can change the implementation of one component, without having to modify all the other components in the system, and one can add functionality to a component, without breaking the behavior of other components that depend on it.

- *Reuse.* Components that prove successful in one system can be used in other systems, in which one would like to obtain comparable behavior.

All these aspects are found in Simon's essay. Similarly, the book *Design Rules: Volume 1. The Power of Modularity* [9] is about the design of computer hardware, and characterizes modularity as "a particular design structure, in which parameters and tasks are dependent within units (modules) and independent across them."

> The concept of modularity spans an important set of principles in design theory: design rules, independent task blocks, clean interfaces, nested hierarchies, and the separation of hidden and visible information. Taken as a whole, these principles provide the means for human beings to divide up the knowledge and the specific tasks involved in completing a complex design or constructing a complex artifact. [9, pp. 89–90]

The authors go on to explain that "modularity does three basic things" that designers might judge to be desirable:

> *1. Modularity increases the range of "manageable" complexity. It does this by limiting the scope of interaction between elements or tasks, thereby reducing the amount and range of cycling that occurs in a design or production process.* As the number of steps in an interconnected process increases, the process becomes increasingly difficult to bring to successful completion....
>
> *2. Modularity allows different parts of a large design to be worked on concurrently.* The independent blocks in a modular task structure can all be worked on simultaneously....
>
> *3. Modularity accommodates uncertainty.* The defining characteristic of a modular design is that it partitions design parameters into those that are visible and those that are hidden. Hidden parameters are isolated from other parts of the design, and are allowed to vary. (Sometimes their range is limited, but they can vary within the range.) Thus from the perspective of the architects and the designers of other hidden modules, hidden parameter values are uncertain. They lie inside a black box known as "the module." [9, pp. 90–91]

Applied to the design of mathematical resources, these goals are appealing: we would like our mathematics to be comprehensible, reliable, flexible, and reusable, and, of course, mathematical contributions are made by agents working independently, at different times and in different locations. Our task in Section 4 will be to understand how the modular design of mathematical artifacts supports these goals.

## 3.2 Modularity in software engineering

The gospel of modular design is most keenly felt in computer science and software engineering. The digital microprocessors that lie at the heart of modern computers embody fairly simple models of computation: they maintain internal registers, move information from memory, carry out arithmetic comparisons, and, importantly, branch on the results of these comparisons to different parts of the code. Long sequences of instructions written in assembly language, which directly represent a machine instruction set, are generally hard to understand and difficult to write, and they are likely to contain mistakes. Although programming languages like Fortran (introduced in the 1950's) and Basic (introduced in the early 1960's) were an advance over assembly language, early programmers still produced long sequences of instructions and branch (*go to*) statements that often resulted in "spaghetti code."

Early programming languages did, however, provide the ability to write subroutines, procedures that could be separated from a block of code and called as though executing a single instruction. This allowed programmers to divide

complex tasks into smaller ones that could be designed and tested independently. In the 1960's, programming methodologies evolved to support a style of implementation wherein a subroutine can be viewed as an independent *module,* conceptually distinct from other parts of the program. Interactions with other pieces of code were mediated by the module's *interface,* which specified the input expected by the subroutine, the output it would return, and any behavior that might alter the global *state* of a computation, which is visible to outside code. A programmer could then focus on writing code in such a way to meet this specification, while other programmers could use the subroutine knowing only the specification, without knowing or caring about the implementation details.

In the 1970's, programming methodology itself became an object of study. At the start of the decade, Niklaus Wirth, designer of the *Pascal* programming language, published a paper titled "Program development by stepwise refinement" [41] in which he considered "the creative activity of programming . . .as a sequence of design decisions concerning the decomposition of tasks into subtasks and of data into data structures." It drew a sequence of four conclusions, the first two of which are as follows:

> 1. Program construction consists of a sequence of refinement steps. In each step a given task is broken up into a number of subtasks. Each refinement in the description of a task may be accompanied by a refinement of the description of the data which constitute the means of communication between the subtasks. Refinement of the description of program and data structures should proceed in parallel.
> 2. The degree of modularity obtained in this way will determine the ease or difficulty with which a program can be adapted to changes or extensions of the purpose or changes in the environment (language, computer) in which it is executed.

In 1972, David Parnas, a member of the Department of Computer Science at Carnegie Mellon University, wrote an influential paper, "On the criteria to be used in decomposing systems into modules" [32]. It begins by quoting a 1970 textbook:

> A well-defined segmentation of the project effort ensures system modularity. Each task forms a separate, distinct program module. At implementation time each module and its inputs and outputs are well-defined, there is no confusion in the intended interface with other system modules. At checkout time the integrity of the module is tested independently; there are few scheduling problems in synchronizing the completion of several tasks before checkout can begin. Finally, the system is maintained in a modular fashion; system errors and deficiencies can be traced to specific system modules, thus limiting the scope of detailed error searching.

Parnas continued:

> The major advancement in the area of modular programming has been the development of coding techniques and assemblers which (1) allow one module to be written with little knowledge of the code in another module, and (2) allow modules to be reassembled and replaced without reassembly of the whole system. This facility is extremely valuable for the production of large pieces of code...

Comparing two ways of breaking a particular program into modules, Parnas argued that the more effective division is one that incorporates "information hiding":

> We propose ...that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others.

The paper also summarizes the reasons to adopt such an approach.

> The benefits expected of modular programming are: (1) managerial—development time should be shortened because separate groups would work on each module with little need for communication; (2) product flexibility—it should be possible to make drastic changes to one module without a need to change others; (3) comprehensibility—it should be possible to study the system one module at a time. The whole system can therefore be better designed because it is better understood.

The word "encapsulation" is often used in place of "information hiding." In 1974, in an essay called "On the role of scientific thought" (eventually published as [19]), the computer scientist Edsger Dijkstra, who was also cited in Parnas' paper, used the phrase "separation of concerns." This phrase has also come to stand as the goal of a modular structuring. In 1978, Glenford Myers, then a software engineer at the IBM Systems Research Institute in New York, wrote a textbook [31] that codified the modular approach and again emphasized the same benefits: understandability, maintainability, flexibility, and reuse.

It was not long before this advice made its way into the undergraduate curriculum. The influential MIT textbook, *Structure and Interpretation of Computer Programs* [1], was first published in 1985; its first three chapters are titled "Building Abstractions with Procedures," "Building Abstractions with Data," and "Modularity, Objects, and State." Most software engineering textbooks today focus on compositional design, and explicitly emphasize the benefits of modularity. The overall message can be summarized as follows:

- Large programs should be divided into independent modules.
- A *module* is a body of code with a well-defined *interface*. The interface specifies what procedures the user can call from the outside, what data these procedures expect, what data these procedures return, what state information the module keeps track of, and how procedural calls change the state.

- The internal workings of the code can otherwise largely be ignored; in particular, code that interacts through the interface is guaranteed to work even if the implementation changes.

This is essentially an instantiation of the notion of a modular system, as characterized in Section 3.1, to the case of software design.

In Section 4 we will regard mathematical artifacts from such a perspective, and consider a piece of mathematics such as a theory or a proof to be modular if it is structured as a collection of components with well-defined interfaces that hide implementation details when possible. We will then consider ways that such a structuring confers comprehensibility, reliability, flexibility, and reuse, just as it does in software design.

## 3.3 Refactoring

The dicta of modularity recommend designing software in certain ways. But large software projects tend to grow and evolve over time, often in haphazard and unpredictable ways, and despite their best intentions teams of software engineers often find the complexity of a body of code getting out of hand. When that happens, it is generally deemed to be a good idea to try revise the code, reorganizing and rewriting various parts, in order to restore modularity and its benefits. In that respect, computer scientists and engineers speak of *refactoring*. Like the sailors on Neurath's boat, they have the task of revising and improving the code while it is still actively being used.

A 1999 textbook by Martin Fowler, *Refactoring: Improving the Design of Existing Code*, describes the methodology. The following passage, contributed by Kent Beck, conveys the central idea:

> Programs have two kinds of value: what they can do for you today and what they can do for you tomorrow. Most of the times when we are programming, we are focused on what we want the program to do today . . .
>
> . . .you know what you need to do today, but you're not quite sure about tomorrow. Maybe you'll do this, maybe that, maybe something you haven't imagined yet.
>
> I know enough to do today's work. I don't know enough to do tomorrow's. But if I only work for today, I won't be able to work for tomorrow at all.
>
> Refactoring is one way out of that bind. . .
>
> Refactoring is the process of taking a running program and adding to its value, not by changing its behavior but by giving it more of these qualities that enable us to continue developing at speed.

The book is a journeyman's guide to restructuring code, reorganizing data, improving interfaces, and improving encapsulation.

When one considers the history of mathematics, one sees that mathematical developments — definitions, proofs, and theories — are often revised, recast, and restructured over time. This can happen on the scale of centuries. In Section 4, however, we will note that the pressures to do so, and the attendant benefits, are similar to the ones involved in refactoring software. Thus viewing historical developments in these terms can help us understand them better.

### 3.4 Characterizing modularity of programs

Textbooks in computer science typically describe modularity without offering a precise definition of the notion. Toward obtaining better formal models of modularity in mathematics, however, it will be helpful to gain additional clarity as to what the concept entails. As noted in Section 3.1, talk of modularity presupposes notions of dependence, interface, and encapsulation. The aim of this section is to better understand the way these notions play out in the setting of computer science.

At face value, pronouncements about modularity of code are precisely that: ascriptions of properties to the syntactic strings of symbols that constitute computer programs. To some extent, it may be possible to make sense of the modularity of something more abstract than a computer program; for example, it may make sense to talk about the modularity of an algorithm, independent of the programming language and the particular piece of code that implements it. But finding an appropriate level of abstraction is likely to be delicate, and not essential to my present goals. So, at least for the time being, it makes sense to start with syntax.

Expressions in a programming language can be used not only to define programs themselves, but also to declare data types and data. Here are some examples, in a made-up programming language:

```
struct point := {xval : float, yval : float}

const pi : float := 3.1415

def gcd (x y : nat) : nat :=
  if y = 0 then x else gcd y (x mod y)

def circle_area (r : float) : float := pi * r^2

def distance (a b : point) : float :=
  sqrt ((a.xval - b.xval)^2 + (a.yval - b.yval)^2)
```

Each of these is a definition, which associates an identifier, the *definiendum*, to an expression, the *definiens*. In the examples, the identifiers that are introduced are `point`, `pi`, `gcd`, `circle_area`, and `distance`. In each case, the expression after `:=` provides the *definiens*.

In addition, each definition either implicitly or explicitly singles out the type of object being defined. Specifically, the first example declares a new data type,

`point`, which is a structure that consists of two integers, denoted `xval` and `yval`. The second example declares `pi` to be a floating point constant; the expression `float` after the colon specifies the type of object that is being defined. The next three examples, `gcd`, `circle_area`, and `distance`, are functions. The parenthesized expressions that appear after the name but before the colon indicate the type of inputs each function expects, whereas the expressions `nat` and `float` after the colon specify the type of output. The first of these is a recursive definition of a function that computes the greatest common divisor of two natural numbers; the next computes the area of a circle with radius `r`, and the final one computes the distance between two points.

Now notice that the body of a definition can make use of identifiers for other objects and data types. These identifiers may be defined in the same file, or imported from another file or library, or built into the system at a fundamental level. For example, the definition of `point` presupposes that the system knows what a `float` is; the definition of the function `circle_area` makes use of `pi`, the multiplication symbol, the exponentiation symbol, and the constant symbol `2`; and the definition of `distance` uses, among other things, `point`, `sqrt`, and the projections `xval` and `yval`, which return the components of a `point`.

This induces a bare-bones notion of syntactic dependence: one definition depends on another if the declaration of the first — the *definiens* and its data type specification — references the *definiendum* of the second. Thus, `circle_area`, for example, depends on `pi`, multiplication, exponentiation, 2, and the `float` data type.

There are other notions of dependence, which may be closer to one's specific concerns. These include:

- *Syntactic correctness.* The syntactic correctness of a definition depends on types of the definitions it refers to. For example, the syntactic correctness of the function `circle_area` depends on the fact that the function `sqrt` expects a floating point input and returns a floating point output.

- *Semantics.* The intended denotation of a definition depends on the denotations of the definitions it refers to. For example, if we take the semantic denotation of a function identifier to be a function from inputs to outputs, the function that a definition denotes depends on the semantic denotations of the identifiers it involves.

- *Semantic properties.* The properties of the object denoted, such as the fact that `circle_area` always returns a nonnegative number, depends on properties of the definitions it depends on, such as the property that `pi` is a positive number.

These can all be taken to be derivative of the notion of syntactic dependence. In other words, each of these kinds of dependence follows from the brute syntatic dependences between program elements.

What counts as the notion of an interface, however, seems to be more sensitive to context. From the point of view of syntactic correctness, it may be

sufficient to think of the interface as being the syntactic specification of the data type: the interface to `point`, for example, specifies that it is a structure with the two projections, `xval` and `yval`). From the semantic point of view, the interface may be the denotation itself: all we need to know to determine the denotation of a defined function are the denotations of the components, independent of how they are implemented. Finally, when it comes to reasoning about properties of the objects that the identifiers denote, the interface may simply be the list of relevant properties. For example, for some purposes, we may only need to know that `circle_area` returns a nonnegative number, or that `gcd` is nonzero if both of its inputs are. In that case, those properties can be included in a formal or informal specification, which then becomes the relevant interface.

What is *encapsulated* is then everything that is left out of the interface. From the point of view of checking syntactic correctness, all that is important is that `pi` denotes a `float`; the particular value is hidden to the definition that references it. From the point of view of determining the denotation of `circle_area`, all we need to know is that `sqrt` computes a certain approximation to the square root function; the details of the definition of that function are again, left hidden. When reasoning about properties of the denoted objects, if we need to know that `circle_area` returns a nonnegative number, then in that context the specification can hide any additional information about the value that is computed.

When we transfer the notions to mathematical definitions and proofs in Section 4, we will see that in a sense some of the issues are more cleanly expressed there. Formal languages used by contemporary interactive proof assistants provide means to define mathematical objects and, moreover, to reason about their properties. As a result, the distinction between interfaces that express data types and interfaces that express properties is not sharp; mathematical interfaces can specify both uniformly.

Let me add a few observations that will be relevant to the discussion of modularity in mathematics. First, notice that we can distinguish between direct and indirect dependencies. One definition may refer to another, which, in turn, refers to another. A definition then depends *directly* on the definitions it refers to in its *definiens*, and indirectly to the ones that occur downstream. In discussions of modularity, it is generally the direct dependencies that we care about, the ones that the definition itself "sees." The whole point to modularization is to organize matters so that the lower level dependencies are managed through the intermediaries.

Second, large programs and libraries tend to be hierarchical in nature. Complex procedures are implemented in terms of simpler ones, and even within the body of a function definition, tasks and steps are often decomposed into blocks. Libraries of routines are often grouped into modules, which are groups of procedures that share data, representations, and supporting utilities.

Concomitant with this hierarchical organization, objects usually come with a well-defined *scope*, which is to say, identifiers are only visible at some points in a development. For example, a local variable `x : nat` may be used within

a function definition, proving a reference that is only defined within the scope of that definition. Or a library module may define utility routines that are only visible to other functions and procedures in the module. This is a way of enforcing separation of concerns and limiting dependence. It is often useful to invoke the notion of a *context*, which one can think of a record of the objects that are visible in a given scope.

Finally, it is important to recognize that in programming languages, dependencies are sometimes left implicit, and ambiguous expressions are sometimes disambiguated by the surrounding data. For example, in the definition of `circle_area`, the multiplication symbol denotes multiplication of floating point numbers, whereas in other situations, it may denote multiplication of integers. Or, when multiplying a floating point number by an integer, the system might insert an implicit *cast*, in this case, the function which converts the integer to a float. In that case, the code may be said to depend on the cast, even though it is not explicitly present in the definition.

This last feature is much more pronounced in ordinary mathematical definitions and proofs, where a tremendous amount of information is left implicit. In an ordinary proof, explanations as to why a certain claim follows from the ones previous to it are often omitted entirely, leaving it to the reader to fill in the justification. Thus we not only have to deal with implicit dependencies of proofs on facts, but also the complexity of filling in these justifications, and the mechanisms that make it possible to do that efficiently. We will return to this issue below.

# 4 Modularity in mathematics

## 4.1 From programs to proofs

Replace "software" by "piece of mathematics" everywhere in the last section, and many of the statements still make sense. Developing mathematics in a modular way should make the mathematics easier to understand, less error-prone, and more flexible and reusable. Our goal now is to explore the analogy and make it more precise.

At least some computer scientists have made this analogy explicit. Part XVII of Robert Harper's *Practical Foundations for Programming Languages* [24] is titled "Modularity," and Chapter 44, "Type Abstractions and Type Classes," opens with the following observation:

> Modularity is not limited to programming languages. In mathematics the proof of a theorem is decomposed into a collection of definitions and lemmas. Cross-references among lemmas determine a dependency structure that constrains their integration to form a complete proof of the main theorem. Of course, one person's theorem is another person's lemma; there is no intrinsic limit on the depth and complexity of the hierarchies of results in mathematics. Mathematical structures are themselves composed of separable parts, as,

for example, a Lie group is a group structure on a manifold.

We have already seen that data type and function type specifications in programming languages can be seen as a way of supporting modularity, providing interfaces that specify how a particular data or function can be used. This discipline is central to Harper's book.

The analogies between mathematical texts and computer programs are fairly straightforward. Mathematical proofs are decomposed into definitions and lemmas, just as programs are decomposed into smaller blocks of code. Ordinary mathematics imposes an interface that regulates talk of Lie groups and complex numbers and encapsulates the specifics as to how these are defined, just as a modular programming style imposes an interface on data structures that encapsulates the details of the implementation.[3]

To be sure, there are differences between writing a program and proving a theorem. One difference lies in the scope of the theorem-proving enterprise: most programs are fairly self-contained, whereas a mathematical theorem can rely on definitions and facts introduced by countless others, over a course of decades or centuries. We would thus expect to see in mathematics the kind of refactoring that occurs in large software projects, for example, with industrial programs that are the product of multiple contributors over a long period of time. And, indeed, we do: in the history of mathematics, it is often the case that concepts are introduced, a theorem is proved, the concepts then are refined, and the proofs are rewritten to improve comprehensibility, robustness, and reusability. This gives hope that programming methodology can help illuminate the way that mathematical theories and proofs evolve.

In a branch of computer science known as *formal verification*, one can now use computational proof assistants to verify the correctness of mathematical theorems, and the formal languages they use will help us solidify the correspondence between mathematical texts and proofs. Working interactively with such a proof assistant, users provide input in stylized proof languages, providing enough information for the system to construct a fully detailed proof in an underlying formal axiomatic system. We can think of such a *proof script* as providing instructions to the system as to how to construct the desired proof. In other words, a proof script is really a program, of sorts. Indeed, practitioners often refer to proof scripts informally as "code."

Interactive theorem proving thereby provides a useful intermediary. Insofar as the texts acted on by computational proof assistants are like computer code, we can speak of modularity of these formal texts in ways similar to the ways we speak of modularity of code. And insofar as these formal texts model informal mathematical language, we can expect that modularity of the formal texts should tell us something about modularity in informal mathematics. Reading

[3]I am grateful to David Waszek for pointing out that Bourbaki discussed aspects of the modular structure of mathematics, though note in those terms (indeed, long before the term "modularity" was widely used). Their manifesto [12] describes the use of axiomatic and structural methods to manage complexity, render mathematics intelligible, and unify different parts of the field. It also emphasizes the resulting gains in economy and efficiency of thought.

an informal mathematical proof and assessing its correctness requires us to keep track of local data and hypotheses, and combine then with background knowledge drawn from a wide variety of domains. The computational verification of a formal proof requires the same. We can therefore optimistically expect that mechanisms for developing formal mathematical theories in a modular way will illuminate the methods we use to develop informal mathematical theories in a modular way, and the benefits of modularity in formal mathematical texts should tell us something about the benefits of modularity in ordinary mathematics.

The analogies between formalized mathematics and software are have also been made explicit in the past. For example, the formalization of the Feit-Thompson Odd Order Theorem, an important first step in the classification of finite simple groups, was a milestone achievement in interactive theorem proving. The project, led by Georges Gonthier, was a joint venture between the French computer science agency Inria and Microsoft Research, Cambridge, and made use of an interactive proof assistant called *Coq*. The project was completed in 2012, and is described in a report written by 14 authors (myself among them) [23]. Even the name of the project, *Mathematical Components*, invokes a catchphrase, "software components," that is used to describe modular programming methodology in software engineering. At the time of writing, an Inria web page[4] describes the project in the following way:

> The object of this project is to demonstrate that formalized mathematical theories can, like modern software, be built out of components. By components we mean modules that comprise both the static (objects and facts) and dynamic (proof and computation methods) contents of theories.

The report on the formalization [23] invokes similar analogies to software design:

> . . .the success of such a large-scale formalization demands a careful choice of representations that are left implicit in the paper description. Taking advantage of Coq's type mechanisms and computational behavior allows us to organize the code in successive layers and interfaces. The lower-level libraries implement constructions of basic objects, constrained by the specifics of the constructive framework. Presented with these interfaces, the users of the higher-level libraries can then ignore these constructions. . .

And later:

> A crucial ingredient [in the success of the project] was the transfer of the methodology of "generic programming" to formal proofs. . .[T]he most time-consuming part of the project involved getting the base and intermediate libraries right. This required systematic consolidation phases performed after the production of new material. The

[4] `http://www.msr-inria.fr/projects/mathematical-components-2/`

> corpus of mathematical theories preliminary to the actual proof of the Odd Order theorem represents the main reusable part of this work, and contributes to almost 80 percent of the total length. Of course, the success of such a large formalization, involving several people at different locations, required a very strict discipline, with uniform naming conventions, synchronization of parallel developments, refactoring, and benchmarking...

The analogy, then, is at least suggestive. Just as we clarified the notion of modularity in programming languages in Section 3.4 with reference to a made-up programming language, let us try to spell out the relevant notions of dependence and interface with respect to a formal proof language.[5]

## 4.2 Toward a formal model

We have seen that in a conventional programming language, identifiers can refer to at least two different sorts of objects: data type specifications, such as `nat`, `float`, and `point` in the examples in Section 3.4, and data itself, such as `pi` and `circle_area`. Notice that I am not distinguishing between constants and functions in treating both as data: the constant `pi` is an object of type `float` and the function `circle_area` is an object type `float` $\rightarrow$ `float`, where the arrow is used to denote a *function type.* In other words, if we think of the function specification as a data type, we can view constants and functions uniformly as data, whose intended usage and behavior are specified by their associated type.

Interactive theorem proving adds two more components to the mix: in addition to specifying mathematical objects and their types, one can also make assertions and prove them. Thus, the language of a proof assistant will provide means to construct expressions denoting all of the following objects:

- data type specifications
- mathematical objects of these types
- propositions
- proofs of these propositions

This list is not meant to be exhaustive: many interactive theorem provers also provide means to organize information and import objects into the current context, configure automation, provide heuristic hints, write new proof procedures, evaluate expressions, and so on. But entities listed above are essential, and it is hard to imagine anything that might be called a theorem prover that does not provide means to describe them.

For illustrative purposes, I will adopt a logical framework known as *dependent type theory*, which provides a single uniform language in which one can

[5] For this purpose, I will in fact use an actual proof language, namely, that of the *Lean* theorem prover [17].

define all four sorts of objects. The use of dependent type theory is not essential to the account, but it is convenient, especially because it also allows dependencies between objects of the different categories. In dependent type theory, there are expressions, and every expression has a type. The novelty is that data types themselves are expressions in the language, which happen to have the type `Type`. Propositions are also expressions in the language, having the type `Prop`. And if `p` is a proposition in the language, a proof of `p` is nothing more than an expression having type `p`. In other words, all four objects above are given by expressions in the same language:

- A data type specification, $\alpha$, is given by an expression of type `Type`.
- A mathematical object of that type is given by expression of type $\alpha$.
- A proposition, `p`, is given by an expression of type `Prop`.
- A proof of that proposition is given by an expression of type `p`.

As in Section 3.4, we can write `e : α` to indicate that expression `e` has type $\alpha$, which in turn determines what sort of object `e` is.

- If `α : Type`, then $\alpha$ is a data type. In that case, `e : α` means that `e` denotes an object of that type.
- If `p : Prop`, then `p` is a proposition. In that case, `e : p` means that `e` is a proof of `p`.

As in Section 3.4, we can also use the general pattern `i : α := e` to denote that the identifier `i` denotes the object of type $\alpha$ defined by `e`, where $\alpha$ can be either `Type`, a particular data type, `Prop`, or a particular proposition. This provides us with a uniform language for expressing data types, objects, assertions, and proofs. To repeat, the use of dependent type theory is not essential here; we could have used four separate languages instead. What is important for the model of mathematical language we adopt here is that (1) we can express all four sorts of objects; (2) every expression has a syntactic type, which indicates what sort of entity it is; and (3) the various syntactic categories interact with one another, as described below.

As examples of types and objects, $\mathbb{N}$ denotes the type of natural numbers, and `bool` denotes the type of Boolean values (`tt` and `ff`, for "true" and "false"). The type $\mathbb{N} \times$ `bool` is the type of pairs consisting of a natural number and a boolean, and the type $\mathbb{N} \to (\mathbb{N} \to \mathbb{N})$ is the type of functions which take two natural numbers as arguments, and return a natural number; here the convention is that the arrow operation associates to the right. In contrast, the type $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ is the type of functionals which take a function from natural numbers to natural numbers as arguments, and returns a natural number.

We can specify variables of these types:

```
variables (m n: ℕ) (f : ℕ → ℕ) (p : ℕ × ℕ)
variable  g : ℕ → (ℕ → ℕ)
variable  F : (ℕ → ℕ) → ℕ
```

Once that is done, `f n`, $\texttt{pr}_1$ `p`, `m + n^2 + 7` are all terms of type $\mathbb{N}$. Note that function application is written without parentheses, so that `g m` is a function of type $\mathbb{N} \to \mathbb{N}$ and `g m n` is an expression of type $\mathbb{N}$. Thus we can view `g` as a function that takes two natural numbers as arguments and returns a natural number, and `F (g m)` is also an expression of type $\mathbb{N}$.

We can write propositions using quantifiers and connectives in the usual ways. For example, consider the following proposition:

```
∀ α : Type, ∀ x y z : α, x = y → y = z → x = z
```

This expresses that for every type $\alpha$, the equality relation on $\alpha$ is transitive. Notice that here we do not have to specify that `=` denotes the equality relation on $\alpha$, since that can be inferred from the fact that the arguments have type $\alpha$.

We can then start writing definitions and proving theorems, which amounts to introducing identifiers to name the various kinds of objects. The following example illustrates this.

```
def binary_relation (α : Type) : Type := α → α → Prop

def transitive {α : Type} (r : binary_relation α) : Prop :=
∀ {x y z}, r x y → r y z → r x z

def binary_relation_inverse {α : Type}
  (r : binary_relation α) : binary_relation α :=
λ x y, r y x

theorem transitive_binary_relation_inverse {α : Type}
    {r : binary_relation α} :
  transitive r → transitive (binary_relation_inverse r) :=
assume h : transitive r,
assume x y z : α,
assume h₁ : binary_relation_inverse r x y,
assume h₂ : binary_relation_inverse r y z,
show binary_relation_inverse r x z,
  from h h₂ h₁
```

The first definition, `binary_relation`, defines a new data type: for every type $\alpha$, `binary_relation` $\alpha$ is the type of binary relations on $\alpha$. Notice that we can represent such a relation as a function `r` which takes two elements of $\alpha$ and returns a proposition. The second definition, `transitive`, introduces a new predicate on binary relations: if `r` is a binary relation on a type $\alpha$, the expression `transitive r` represents the assertion that `r` is transitive. The curly brackets in the definition specify that we do not need to indicate the underlying type $\alpha$ explicitly, since it can be inferred from the type of `r`; in other words, we can write `transitive r` instead of `transitive` $\alpha$ `r`, thereby leaving the dependence on $\alpha$ implicit. (These implicit dependencies were foreshadowed in Section 3.4, and are discussed in further detail below.) The function `binary_relation_inverse` takes a binary relation `r` as input, and returns the

inverse relation: `binary_relation_inverse r x y` holds if and only if `r y x` holds. Finally, `transitive_binary_relation_inverse` names the theorem that if a binary relation `r` is transitive, so is `binary_relation_inverse r`. The expression following `:=` is a proof of that theorem.

The precise syntax of dependent type theory need not concern us here. What is important is that, as in Section 3.4, the association of identifiers to expressions induces a notion of dependence: an expression depends on the identifiers it mentions. For example, the definitions of transitivity of the inverse relation, presented above, depend on the notion of a binary relation. The theorem `transitive_binrel_inverse` depends, in turn, on the depend on the notions of transitivity and inverse relation. The proof of `transitive_binrel_inverse` uses nothing beyond pure logic, but if the proof invoked other lemmas, theorems, and constructions, we would have a formal record of those dependencies as well.

Recall that in Section 3.4, we observed that there are derivative notions of dependence associated with the semantic reference of the expressions involved. Here, however, there is less of a need to invoke semantic notions. Since interactive theorem provers rely on a foundational language to specify all mathematical objects and their properties, it is not clear that there is anything to be gained by stepping outside the system and worrying about semantic reference. Similarly, because we can assert and establish facts about the objects we define within the foundational language, dependencies between properties are tracked by syntactic references as well.

The examples make it clear that an expression of one sort can depend on entities of other sorts. For example, the definition of a mathematical object can depend on other objects and data types, and a proof can depend on data types, objects, propositions, and other proofs. Some of the dependencies that can occur are not as obvious. The expression `if even x then 0 else 1` denotes an object (in this case, a natural number), but it depends on the proposition `even x`. More strikingly, the definition of an object can depend on a proof; if we were to define `gcd x y` as the greatest common divisor of `x` and `y`, we would have to provide a proof that this description characterizes a unique object.

As noted in Section 3.4, what we generally care about are the direct dependencies between expressions and identifiers, but some dependencies may be implicit. Expressions in interactive theorem provers often elide information which is inferred and inserted by the system. For example, the system may infer that a multiplication symbol denotes multiplication in a group; in that case, the expression implicitly depends on the notion of a group, and on the notion of multiplication in a group. The situation is even more complicated with proofs. Ordinary mathematical proofs often omit detailed justifications and leave it to the reader to fill in the details. This is mirrored in an interactive theorem prover by the fact that often proofs are supplied by automated routines, which invoke theorems and constructions that are invisible to the user. In that case, we should say that the surface proof implicitly depends on the facts and data invoked by the automation; or in some contexts, perhaps, it would be more illuminating to say that the proof depends on the steps supplied by the

automation, treating those steps as black boxes.

## 4.3 Mathematical interfaces

Given the centrality of the notion of an interface in computer science, we should now say something about how it plays out in a mathematical setting. In Section 3.4, we saw that the notion of interface is slippery and context dependent; what is considered an interface in computer science can depend, for example, one whether one is trying to account for syntactic correctness, the denotation of a program, or specific properties. It can also depend on the object of analysis, which can be a single function, data structure, or procedure, or a module or library that bundles a number of these together to provide useful functionality.

The same is true in the mathematical setting. To start with, as was the case with computer programs, type information can be viewed as an interface for mathematial objects and functions. Knowing that an expression `e` has type $\mathbb{N}$ means that one can profitably write `e + 7` and send it to other functions that expect a natural number as input. It also specifies that it can serve to instantiate any theorem that makes a general statement about natural numbers. Similarly, knowing that an expression `f` has type $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$ means that it can be applied to two natural numbers to obtain a natural number, and also that it can be send as an argument to another function that expect such a function as an argument.

Analogously, the "interface" to a theorem-proof pair is the statement of the theorem itself. Suppose we have a proof of Fermat's last theorem in our library:

```
theorem fermat :
  ∀ x y z n : ℕ, n > 2 ∧ x * y * z ≠ 0 → x^n + y^n ≠ z^n :=
  ...
```

The statment of the theorem specifies that it can be applied to any tuple of natural nunmbers `x`, `y`, `z`, and `n`, provided `n > 2` and `x`, `y`, and `z` are not all zero. The very act of stating and proving a theorem presents a powerful form of encapsulation: anyone can make use of the theorem knowing only the statement of the theorem and the fact that it has been proved. The details of the proof can remain hidden. Mathematics would be unworkable if we had to recapitulate the proof of a theorem each time we want to use it, and so this type of encapsulation is essential to the reusability of theorems and the ability of different mathematical communities to develop results independently and share them after the fact.

But if we try to transfer this observation back to expressions that denote objects and functions, we find that the analogy breaks down: the type of an object or function is clearly *not* sufficent to specify all aspects of its proper use. Knowing that a function has type $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$ tells us that it expects two natural numbers as arguments and returns a natural number, but it doesn't tell us anything more than that: the function may be addition, or multiplication, or it may return the greatest common divisor of its inputs.

Even when it comes with definitions of objects and functions, however, encapsulation plays an important role. Foundationally, there are many ways of defining the real numbers. For example, they can be defined as equivalence classes of Cauchy sequences or as Dedekind cuts. For most purposes, the specific choice is irrelevant, and conventional textbooks are usually entirely agnostic as to how they are defined. Along these lines, most theorem provers provide mechanisms to choose which aspects of a formal library to make publicly available and which to hide from view. For example, a library for the real numbers might expose arithmetic operations on the real numbers and their basic properties while hiding the specific details of how the reals are implemented. In that sense, the body of publicly available theorems and functions serve as the interface to the library.

One way mathematics manages such interfaces is to encode them as algebraic structures. The real numbers can be characterized uniquely, up to isomorphism, as a complete archimedean ordered field. The algebraic structure known as an *ordered field* specifies a signature of functions and relations that any instance must implement, and the properties that they must satisfy. Theorems can then be proved generically for any structure that meets that specification. Instantiating the real numbers as an ordered field makes those theorems available in that particular instance. In that sense, ordered ring structure provides an interface to the real numbers (as well as to the integers and rationals).

The situation may seem disappointing: we are looking for a notion of interface in mathematics, and now we have an unruly host of candidates on offer. What does this say about our attempts to discern modular structure in mathematics?

It should be encouraging that computer science fares no better in this regard. For all the talk of modularity and interfaces, there is no univocal interpretation of the term in that field. Depending on the context, computer scientists make speak of an interface to a particular data structure or function, an interface to a collection of data types and functions bundled together into an object or module, or an interface to a complex system or body of code. This does not seem to be a handicap. They can be very precise about particular mechanisms that support modularity, while allowing the notion of interface to remain fluid.

We should expect the same to be the case for mathematics, whether we analyze it in formal or informal terms. Information hiding, encapsulaton, and interfaces are important to mathematics, but what is being hidden and encapsulated can vary depending on context, and different mechanisms are used to make it happen. It will not help us to impose an artificial order. We have to analyze the data as we find it, and try to obtain a better understanding of the way that mathematical information is managed effectively.

I have so far argued that formal methods and interactive theorem proving provide us with a conceptual scaffolding that can help us make sense sense of modularity in mathematics and understand how it plays out in informal mathematical texts. Section 5 takes some initial steps in analyzing examples from ordinary mathematics in these terms.

## 4.4 Measures of complexity

Maintaining modularity in software is supposed to make code easier to understand, easier to maintain, and easier to extend, and to increase the likelihood that the code can be reused in other contexts. We would like to make the case that maintaining modularity in mathematical theories has similar benefits. Making this case presupposes some conception of what it means to be easier to understand, maintain, or extend a theory. Whether we try to formalize these assessments or deal with them at an informal level, it still behooves us to clarify the measures of understandability, maintainability, or extendability we have in mind. Here I will take only a few small steps in this direction.

Let us focus on the benefits of modularity with respect to mathematical proofs. At least two measures of difficulty come to mind: we can consider how a modular organization makes it easier to *find*, or *discover*, a mathematical proof in the context of a background theory, or we can consider how a modular organization makes it possible for us to read and understand a proof that is given to us. The distinction between the two is not sharp: part of understanding a proof involves being able to fill in justificatory steps, and explain why an assertion follows from previous ones. In other words, part of understanding a proof and verifying its correctness involves rediscovering small chains of reasoning that are left implicit. Since, however, the task of processing an existing proof seems more straightforward than the task of finding a new one *ab initio*, the former seems to be a good place to start.

Even making sense of *that*, however, is not an easy task. As I noted in Section 3.4, what makes mathematical proofs, even formalized ones, different from computer code is the amount of information that is ordinarily left implicit. Reading a proof is a complex task: when we do so, we need to keep track of the objects and facts that are introduced, muster relevant background knowledge, and fill in nontrivial reasoning steps that are nonetheless deemed to be straightforward by the author. We should expect that a modular structuring of the background knowledge, as well as the proof itself, should decrease the cognitive burden in all the following ways:

- Type specifications make it possible for us to infer the types of objects and expressions in front of us, for example, to recognize that one expression denotes a natural number while another denotes an element of some group.

- Types and axiomatic structures make it easier to apply constructions and theorems, telling us exactly what data is necessary and what side conditions need to be dispelled, and giving us the means to recognize the structure that the constructions and theorems presuppose.

- Types, axiomatic structures, and modular structuring of theories makes it easier to find and retrieve relevant facts from our background knowledge, since the background knowledge is organized by topic and key constructions.

- Encapsulation keeps information overload at bay. Rather than require us to keep an overwhelming level of detail fresh in our minds, a modular structuring ensures that we only keep track of the information that is essential to the inferential structure of the proof, suppressing extraneous and distracting details.

Spelling out a precise model to justify these intuitions is no small task. But the concerns are familiar to those who have worked in interactive theorem proving and automated reasoning. These fields provide formal algorithmic descriptions of fundamental cognitive tasks: *matching* refers to methods that make it possible to instantiate a theorem or generic construction to specific data, *unification* refers to methods that, more generally, make it possible to instantiate variables in such a way as to make terms or hypotheses match, and *indexing* refers to methods that make it possible to find relevant data and facts quickly (see, for example, [35]). It is therefore reasonable to seek robust and cogent explanations as to how modularity supports these fundamental tasks.

# 5 Examples from number theory

In informal mathematics, modularity is everywhere you look. Take any textbook off the shelf, and you will find definitions and theorems organized into chapters according to topic, in such a way the later appeals to them are regimented and controlled. Every definition encapsulates information in its definiens, and theorems are carefully designed to manage the way we work with the mathematical objects so defined. Axiomatically defined structures in algebra and analysis provide interfaces to instances thereof.

Nonetheless, considering a few specific examples will be informative, and will help illustrate some of the ways that mathematical definitions and concepts encapsulate information, manage the flow of data, and facilitate reuse.

## 5.1 Congruence

Let us start with an example from number theory, one that is simple but nonetheless illustrates some of the relevant phenomena.

**Definition 5.1.** *If $x$ and $y$ are integers, then $x$* divides *$y$, written $x \mid y$, if there is an integer $z$ such that $y = xz$.*

**Definition 5.2.** *If $x$, $y$, and $m$ are integers, then $x$* is congruent to *$y$* modulo *$m$, written $x \equiv y \pmod{m}$, if $m \mid x - y$.*

The fact that computations modulo an integer $m$ is known as "modular arithmetic" is *apropos*. If we want to determine what day of the week it will be 1,000 days from today, we only care about the remainder upon division by seven, and modular arithmetic provides an interface which abstracts, or encapsulates, any additional information. Here is an example of something that can be proved using these notions.

**Proposition 5.3.** *If $x \equiv y \pmod{m}$, then $x^3 + 3x + 7 \equiv y^3 + 3y + 7 \pmod{m}$.*

Here is a brute-force proof.

*Proof.* Unpacking definitions, we have $x \equiv y \pmod{m}$ if and only if $x = y + mz$ for some $z$. Then

$$\begin{aligned} x^3 + 3x + 7 &= (y + mz)^3 + 3(y + mz) + 7 \\ &= y^3 + 3y^2mz + 3ym^2z^2 + m^3z^3 + 3y + 3mz + 7 \\ &= y^3 + 3y + 7 + m(3y^2z + 3ymz^2 + m^2z^3 + 3z) \end{aligned}$$

which shows that $x^3 + 3x + 7 \equiv y^3 + 3y + 7 \pmod{m}$. □

Of course, this is not the sort of proof one expects to see in mathematics. For one thing, it does not scale well: replace $x^3$ by $x^{30}$ and the calculation becomes unbearable. But what is more notable here is that it breaks an abstraction barrier. The existential quantifier in the definition of "$x$ divides $y$" serves to encapsulate information, namely, hiding the value of $z$ such that $y = xz$. We introduce such a definition precisely for that purpose. Then, when we define congruence in terms of divisibility, we expect properties of the former to be obtained by properties of the latter. The following is a modularization of the above proof that respects that abstraction.

**Proposition 5.4.** *Let $x$, $y$, and $z$ be integers.*

1. *$x \mid x$.*
2. *If $x \mid y$ and $y \mid z$, then $x \mid z$.*
3. *If $x \mid y$ and $x \mid z$, then $x \mid y + z$.*
4. *If $x \mid y$, then $x \mid zy$.*
5. *$x \mid 0$.*

*Proof.* For the first claim, we have $x = x \cdot 1$. For the second claim, if $y = xu$ and $z = yv$, then $z = x(uv)$. For the third claim, if $y = xu$ and $z = xv$, then $y + z = x(u + v)$. For the fourth claim, if $y = xu$, then $zy = x(zu)$. The fifth claim follows from the fact that $0 = x \cdot 0$. □

With Proposition 5.4 in hand, we no longer need to unfold the definition of divisibility. In fact, the proof of Proposition 5.4 is the only place we need to provide explicit witnesses to the existential quantifier.

**Proposition 5.5.** *For a fixed $m$, the relation $x \equiv y \pmod{m}$ is an equivalence relation, which is to say, it is reflexive, symmetric, and transitive.*

*Proof.* Since $m \mid x - x$, we have $x \equiv x \pmod{m}$. If $x \equiv y \pmod{m}$, then $m$ divides $x - y$, and so it divides $-1(x - y)$, which is equal to $y - x$. This implies $y \equiv x \pmod{m}$. To see that congruence is transitive, suppose $x \equiv y \pmod{m}$ and $y \equiv z \pmod{m}$. Then $m$ divides both $x - y$ and $y - z$, and hence it divides their sum, $x - z$, as required. □

**Proposition 5.6.** *1. If $x \equiv y \pmod m$, then $x + z \equiv y + z \pmod m$*

*2. If $x_1 \equiv y_1 \pmod m$ and $x_2 \equiv y_2 \pmod m$ then $x_1 + x_2 \equiv y_1 + y_2 \pmod m$.*

*3. If $x \equiv y \pmod m$, then $xz \equiv yz \pmod m$.*

*4. If $x_1 \equiv y_1 \pmod m$ and $x_2 \equiv y_2 \pmod m$ then $x_1 x_2 \equiv y_1 y_2 \pmod m$.*

*5. If $x \equiv y \pmod m$, then $x^n \equiv y^n \pmod m$ for every natural number $n$.*

*Proof.* For the first claim, $(x + z) - (y + z) = x - y$, so if $m$ divides $x - y$, it divides $(x+z)-(y+z)$. The second identity is obtained by applying the first one twice, using the commutativity of addition and the transitivity of congruence. For the third claim, if $m$ divides $x - y$, then it divides $(x - y)z$ by clause 3 of Proposition 5.4. The fourth claim is obtained by applying the third claim twice, and the last is obtained by induction on $n$, using clause 4. □

It now follows that if $p(x)$ is any polynomial in $x$ with integer coefficients and $x \equiv y \pmod m$, then $p(x) \equiv p(y) \pmod m$. Formally, this can be proved by induction on the number of monomials in $p$. Proposition 5.3 is merely a special case.

This simple example nicely illustrates the way a mathematical definition can suppress information. In this case, it is not a matter of being able to compute the missing data: if $x$ divides $y$, then $y$ is equal to $x(y/x)$. At odds is simply whether $y/x$ is an integer. Our first proof of Proposition 5.3 shows that it is by expressing it explicitly in terms of $x$, $z$, and $m$. In some cases, this information may be useful, but when it is not, keeping it around is a distraction. Our second proof therefore suppresses it. In practice, it is often not clear what information should be hidden and what should be left explicit; these are design decisions that require mathematical judgment, and remain subject to revision as a theory evolves.

Mathematics is replete with information hiding of this sort. In analysis, if $f$ is a function from the real numbers to the real numbers, writing $\lim_{x \to a} f(x) = b$ means that for every $\varepsilon > 0$ there is a $\delta > 0$ with the property that whenever $|x - a| < \delta$, $|f(x) - b| < \varepsilon$. Thus any limit statement encapsulates information, namely, the dependence of $\delta$ on $\varepsilon$. Once again, it is the use of the existential quantifier that serves to hide the relevant data. The notion of a limit is used in later definitions, such as that of continuity, differentiation, and integration, and calculus provides rules for establishing continuity and calculating derivatives and integrals without providing explicit rates of convergence. Here the suppression is less benign: for the purpose of approximating derivatives and integrals numerically, having a bound on the rate of convergence is of utmost importance, and numerical analysis provides means of obtaining these. Conventional theories of analysis, however, suppress quantitative information in favor

of a qualitative understanding of the phenomena involved.[6] In that way, the limit concept is an effective means of information management.

The refactored proof of Proposition 5.3 is not shorter than the original if we count the auxiliary propositions, but those can be reused, and yield a much more general result. And even in this simple case, breaking the proof into small pieces makes each step easier to check and understand, and reduces the risk of error.

## 5.2 Fermat's Little Theorem

For an example of refactoring where the gains in the refactored proof are not solely attributable to the suppression of information, consider the following, known as *Fermat's little theorem.*

**Theorem 5.7.** *Let $p$ be any prime number, and suppose $p \nmid a$. Then $a^{p-1} \equiv 1 \pmod{p}$.*

This fact was known to Fermat, and Euler published a proof in 1761. An excerpt of Euler's proof appears in translation in Struik's sourcebook [39]. Modern terminology agrees with Euler's in using the phrase "the *residue* of $a$ modulo $p$" to denote the remainder upon dividing a number $a$ by $p$. Before the beginning of the text excerpted by Struik, Euler has shown that for any prime $p$ and any $a$ not divisible by $p$, there is a value $\lambda$ such that $a^\lambda \equiv 1 \pmod{p}$. He has also shown that if $a$ is not 1, then for the least such value $\lambda > 0$, the residues of

$$1, a, a^2, a^3, \ldots, a^{\lambda-1}$$

are distinct and not equal to 0. In modern terms, we would say that the Euler has essentially shown that the order of $a$ is $\lambda$ modulo $p$. In particular, $a^{\lambda-1}$ is the multiplicative inverse of $a$ modulo $p$. Since every nonzero residue has an inverse and the product of two nonzero residues modulo $p$ is again a nonzero residue modulo $p$, we have that the set of nonzero residues modulo $p$ form a finite *group* under multiplication modulo $p$. We would also say that the set of residues of $\{1, a, \ldots, a^{\lambda-1}\}$ forms a subgroup of this group, that is, it is a set that is closed under the product operation.

With those results in place, Euler builds to the proof of Theorem 5.7 with a sequence of theorems and corollaries, of which the following is the first. (Here we have corrected a minor typographical error in Struik's translation.)

> **Theorem 5.8.** *If the number of different residues resulting from the division of the powers $1, a, a^2, a^3, a^4, a^5$, etc., by the prime number $p$ is smaller than $p - 1$, then there will be at least as many numbers that are nonresidues as there are residues.*

[6] There is a nice discussion of this in a blog post by Terence Tao, `https://terrytao.wordpress.com/2007/05/23/soft-analysis-hard-analysis-and-the-finite-convergence-principle/`.

*Proof.* Let $a^\lambda$ be the lowest power which, when divided by $p$, has the residue 1, and let $\lambda < p-1$; then the number of different residues will be $= \lambda$ and therefore smaller than $p-1$. And since the number of all numbers smaller than $p$ is $= p-1$, there obviously must in our case be numbers that do not appear in the residues. I claim that there are at least $\lambda$ of them. To prove it, let us express the residues by the terms themselves that produce them, and we get the residues

$$1, a, a^2, a^3, \ldots, a^{\lambda-1},$$

whose number is $\lambda$ and, reducing them in the usual way, they all become smaller than $p$ and are all different from each other. As $\lambda$ is supposed to be $< p-1$, there exists certainly a number not occurring among those residues. Let this number be $k$; now I say that, if $k$ is not a residue, then $ak$ and $a^2k$ and $a^3k$ etc. as well as $a^{\lambda-1}k$ do not appear among the residues. Indeed, suppose that $a^\mu k$ is a residue resulting from the power $a^\alpha$; then we would have $a^\alpha = np + a^\mu k$ or $a^\alpha - a^\mu k = np$ and then $a^\alpha - a^\mu k = a^\mu(a^{\alpha-\mu} - k)$ would be divisible by $p$. Now $a^\mu$ is not divisible by $p$, so $a^{\alpha-\mu}$ would, if divided by $p$, give the residue $k$ contrary to the assumption. From this it follows that all the numbers $k, ak, a^2k, \ldots, a^{\lambda-1}k$ or numbers derived from them are nonresidues. Moreover, they are all different from each other and their number is $= \lambda$; for if two of them, say $a^\mu k$ and $a^\nu k$, divided by $p$ were to give the same residue $r$, then $a^\mu k = mp + r$ and $a^\nu k = np + r$ and thus $a^\mu k - a^\nu k = (m-n)p$, or $(a^\mu - a^\nu)k = (m-n)p$ would be divisible by $p$. Now $k$ is not divisible by $p$, since we have assumed that $p$ is a prime number and $k < p$; then $a^\mu - a^\nu$ would have to be divisible by $p$; or $a^{\mu-\nu}$ would give, divided by $p$, the residue 1, which is impossible because $\mu < \lambda - 1$ and $\nu < \lambda - 1$; also $\mu - \nu < \lambda$. Therefore all the numbers $k, ak, a^2k, \ldots, a^{\lambda-1}k$, if reduced, will be different and their number is $= \lambda$. Thus there exist at least $\lambda$ numbers not belonging to the residues so long as $\lambda < p-1$. □

This is only the first 32 lines of the excerpt, in which the proof of Theorem 5.7 ends on line 127; in other words, the remainder of Euler's proof runs three times as long as the excerpt. Part of the length can be attributed to the fact that Euler makes no effort to be concise. But contemporary proofs also introduce concepts that streamline the presentation, and it will be informative to consider how that works.

Reverting to modern terminology, let $G$ be the group of nonzero residues modulo $p$, and let $H = \{1, a, a^2, a^3, \ldots, a^{\lambda-1}\}$ be the subgroup generated by $a$, where now we take the power operation modulo $p$. If $k$ is any element of $G$, let $Hk$ denote the *coset* $\{hk \mid h \in H\}$, that is, the set of elements of the form $hk$ for some $h \in H$. Notice that $Hk$ is a subset of $G$.

**Proposition 5.9.** *For any $k, r \in G$, if $k \notin Hr$, then $Hk \cap Hr = \emptyset$.*

*Proof.* We prove the contrapositive. If $g$ is an element of the intersection, then $g = h_1 k = h_2 r$ from some $h_1, h_2 \in H$. Multiplying by $h_1^{-1}$ on the left, we obtain $k = h_1^{-1} h_2 r$, which is an element of $Hr$, since $h_1^{-1} h_2 \in H$. □

**Proposition 5.10.** *For any $k$ in $G$, the cardinality of the coset $Hk$ is equal to the cardinality of $H$, that is, $|Hk| = |H|$.*

*Proof.* The map which sends any element $h$ of $H$ to $hk$ is a bijection from $H$ to $Hk$: it is clearly surjective, and if $h_1 k = h_2 k$, then, multiplying both sides by $k^{-1}$ on the right, we have $h_1 = h_2$. □

**Proposition 5.11.** *The cardinality of $H$ divides the cardinality of $G$.*

*Proof.* Let $g_1 = 1$. If $Hg_1$ is not equal to all of $G$, pick an element $g_2$ in $G$ but not $Hg_1$. If $Hg_1 \cup Hg_2$ is not all of $G$, pick an element $g_2$ in $G$ but not $Hg_1$ or $Hg_2$, and so on. Since $G$ is finite, eventually we obtain

$$G = Hg_1 \cup Hg_2 \cup Hg_3 \cup \ldots \cup Hg_n$$

for some sequence $g_1, \ldots, g_n$. We have shown that the sets $Hg_1, Hg_2, \ldots, Hg_n$ are disjoint and each has cardinality $|H|$, so $|G| = |H| \cdot n$. □

Theorem 5.7 now follows: since $|G| = p - 1$ and $|H| = \lambda$, assuming $|G| = |H| \cdot n$ we have

$$a^{p-1} \equiv a^{\lambda n} \equiv (a^{\lambda})^n \equiv 1^n \equiv 1 \pmod{p}.$$

It is often said that the proof I have just given is "implicit" in Euler's proof. The excerpted passage is just the first step in his proof of Proposition 5.11: Euler shows that if $k$ is not an element of $H$, then $H \cup Hk$ has twice as many elements as $H$. An important difference between Euler's proof and the refactored version is that the latter relies solely on properties of the group operations — multiplication and the inverse function — while Euler's calculations rely in the details of this *particular* multiplication. This requires descending to the level of powers of $a$, integer multiplication, the act of taking residues, and properties of congruence modulo $p$. There is no notation for congruence, and Euler does not explicitly use properties of divisibility; rather, the calculations are expressed in terms of the arithmetic operations modulo $p$. As a result, properties that I highlighted at the start of this section as implicit in Euler's earlier proof are replayed in detail in this specific instance.

Notice that calculations in the refactored version of Euler's proof are all carried out via the group interface, which is to say, only generic properties of multiplication and inverses are used, as sanctioned by the group axioms. As a result, Propositions 5.9–5.11 are true of any group $G$ and subgroup $H$, finite or not. Thus it establishes this much more general fact, known as Lagrange's theorem.

**Theorem 5.12.** *Let $G$ be any finite group, and let $H$ be any subgroup. Then the cardinality of $H$ divides the cardinality of $G$.*

In particular, for any element $a$ of $G$, if we let $H$ be the cyclic subgroup generated by $a$, we obtain $a^{|H|} = 1$. This is useful in contexts that have nothing to do with arithmetic, but it also yields a generalization of Fermat's theorem. For any integer $n > 1$, the residues modulo $n$ that are relatively prime to $n$ (that is, share no nontrivial common factor) also form a group, whose cardinality is now denoted $\varphi(n)$. Euler's argument establishes that, more generally, if $a$ is relatively prime to $n$, then $a^{\varphi(n)}$ is congruent to 1 modulo $n$, a fact that Euler made explicit in a paper published two years later, in 1763. The result is now known as *Euler's theorem*, and the function $\varphi$ is now known as the *Euler $\varphi$ function.*

Another feature of the refactored proof is that it takes advantage of set-theoretic language and notation that was not available to Euler, and makes use of general set theoretic properties. For example, we make use of the fact that in order to show that the cardinalities of two sets are equal, it suffices to show that there is a bijection between them. We also make use of the fact that the cardinality of a union of a finite disjoint collection of finite sets is the sum of their cardinalities. These are things that Euler does implicitly, but modern terminology streamlines the argument by providing a clean library and interface to such properties.

In sum, we have once again the expected benefits of a modular development: the individual components of the proof are easier to understand, verify, and adapt to other purposes, and the results are more general, and reusable.

## 5.3 Historical examples

The goal of this section is to gesture toward some episodes in the development of nineteenth century number theory where the effects of modularity can be discerned. Studying the development of number theory is often illuminating, in that there are many problems that can be stated in elementary terms, but whose solutions require substantial mathematical machinery. It is informative to study the way that such machinery – concepts invoked from analysis and algebra, for example — helps tame a difficult problem and make it manageable. Moreover, the historical record provides examples of how proofs are revised and rewritten, with the aim of making them easier to understand, as well as with the aim of generalizing the results. Thus the development of number theory provides excellent examples of refactoring, enabling us to discern the factors that guide the process.

I will briefly discuss four problems in number theory that were present at the turn of the nineteenth century: proving the law of quadratic reciprocity, classifying the binary quadratic forms, proving that there are infinitely many primes in any arithmetic progression in which the first term and common difference are coprime, and determining the asymptotic distribution of the prime numbers. The first two of these were solved by Gauss in his *Disquisitiones Arithmeticae* of 1801, and the third was solved by Dirichlet in 1837. The last problem was not solved until 1896, when Hadamard and de la Vallée Poussin, independently, proved the Prime Number Theorem. Let us briefly consider each of these, in

turn, with an eye toward understanding how notions of modularity can help us make sense of the historical developments.

The law of quadratic reciprocity is an identity that determines whether a prime $p$ is a perfect square modulo another prime $q$ in terms of whether $q$ is a perfect square modulo $p$. Legendre claimed this result in 1785, but there was a gap in his proof which will be discussed below. In the *Disquisitiones*, Gauss pointed out the gap and claimed credit for being the first one to give a complete proof of the result. In fact, he gave two proofs in the *Disquisitiones*, and published four additional proofs during his lifetime. Two more proofs were found in his *Nachlass*. Since then, the aim of obtaining powerful generalizations of the law of quadratic reciprocity has been a guiding theme in the development of modern number theory, and the ability to obtain the law of quadratic reciprocity as an easy consequence of a new theory has been seen to be a mark of success. In an appendix to his book, *Reciprocity Laws* [26], Franz Lemmermeyer enumerated 236 published proofs of the theorem.

Thus the law of quadratic reciprocity is an example of refactoring *par excellence*. Some of the proofs are only minor variants of each other, but the full range exhibits radically different methods and ideas. Gauss' original proof was a brute-force induction that is singularly unilluminating. Some proofs invoke properties of the complex numbers, while others use algebraic or geometric methods. In 1879, Dedekind showed how it could be obtained from his new theory of ideals in an algebraic number field, which now forms a core part of algebraic number theory. Thus the case study provides fertile ground for understanding of how different proofs manage and encode information.

A (binary) *quadratic form* is an expression of the form $ax^2+bxy+cy^2$, where $a$, $b$, and $c$ are integers. A beautiful theorem due to Fermat is that a prime number $p$ other than 2 can be written as a sum $p = x^2 + y^2$ of two squares (that is, $p = x^2 + y^2$, where $x$ and $y$ are integers) if and only if $p$ is congruent to 1 modulo 4. This raises the more general problem of characterizing the primes, and, moreover, all the integers that can be represented by a quadratic form $ax^2 + bxy + cy^2$, in terms of the parameters $a$, $b$, and $c$.

In a *tour de force*, Gauss undertook a classification of binary quadratic forms in the Chapter 5 of the *Disquisitiones*, a chapter that is longer than the other six combined. To that end, Gauss introduced a notion of *composition* of binary forms, and used a long and exceedingly difficult calculation to show that the composition law is associative. Harold Edwards writes:

> ...perhaps the profoundest way in which Section 5 affected the development of mathematics lay in the challenge that it presented. Starting with Dirichlet, and continuing with Kummer, Dedekind, Kronecker, Hermite, and countless others, the unwieldy but fruitful theory of composition of forms called forth great efforts of study and theory-building that shaped modern mathematics. [21, p. 108].

Indeed, the development of the theory can be seen as a long process of refactoring and reconceptualization. Today we interpret Gauss' result as telling us that equivalence classes of binary quadratic forms constitute a group under the

composition law. Dedekind, with his theory of ideals, was able to translate the problem to the study of the *class group*, a group of equivalence classes in an algebraic number field related to the original binary quadratic form. Historical information can be found in Cox, *Primes of the Form $x^2 + ny^2$: Fermat, Class Field Theory, and Complex Multiplication* [16]. The literature on binary quadratic forms is vast, and understanding how various approaches package and manage information can illuminate the strategies that are used to situate a difficult mathematical problem in a broader conceptual framework. Another interesting feature of the history is that although the motivating problem has a computational character, various abstractions pull away from and suppress computational information. Computational theories of binary quadratic forms (see e.g. [13]) aim to recapture algorithmic information, and it is important to understand how the computational theories interact with the conceptual ones. The process of refactoring is still ongoing: quite recently, in fact, Manjul Bhargava has identified Gauss' composition law as an instance of a more general construction [11].

Our third example is Dirichlet's theorem on primes in an arithmetic progression. When Legendre tried to prove the law of quadratic reciprocity, he assumed that there are infinitely many primes in any arithmetic progression $a, a + d, a + 2d, \ldots$ in which $a$ and $d$ have no common factor. He did not prove this claim, however, and this is precisely the gap that Gauss identified in the *Disquistiones*. Gauss was able to circumvent the assumption, but, in fact, he was never able to prove it. It was Dirichlet who managed to do so, in 1837, with a striking approach that combined novel algebraic ideas as well as sophisticated analytic arguments. Rebecca Morris and I have studied the history of subsequent presentations and reformulations of Dirichlet's proof, over a 90-year period, with an eye toward understanding the effects of these reconceptualizations [8, 7]. We show that, indeed, the historical process can be naturally understood in terms of a drive to increase modularity.

Finally, consider the distribution of primes. The density of prime numbers among the first $n$ integers generally decreases as $n$ increases; for example, four among the first ten positive integers are prime, but only 25 of the first 100. At the turn of the nineteenth century, Gauss, on the basis of calculation, conjectured that the number of primes is asymptotic to $n/\log n$ in the limit, which is to say, the ratio of the two quantities approaches 1 as $n$ approaches infinity. In 1859, Bernhard Riemann established a connection between the distribution of primes and the zeros of a complex-valued function now known as the Riemann zeta function. Even with this crucial step forward, the result was not obtained until 1896, when it was proved by Jacques Hadamard and Charles de la Vallée Poussin independently. (There is a nice historical account in [20].)

Because the statement of the theorem involves a limit, the role of analysis in the proof is perhaps not surprising. But the role of the complex numbers is intriguing: as with any abstraction, here the methods of complex analysis serve to encapsulate certain bits of information while making other information salient. Once again, a detailed study of the way the mathematical definitions, theorems, and proofs serve to tame complexity will help us understand how

mathematical abstractions serve to support the reasoning process.

# 6 Conclusions

This exploration of modularity in mathematics has been broad and programmatic, and more detailed work is needed to make the account fully satisfying. Nonetheless, I hope I have provided a framing of some of the issues that bear on the development and normative assessment of mathematical resources that can help orient and guide their study. In this final section, I will indicate some directions for future work, and summarize the central themes of this essay.

## 6.1 Modularity of method

As discussed in Section 2, we can model mathematical practice on two levels. On the one hand, we have the fairly concrete syntactic data, the definitions, theorems, proofs, conjectures, questions, and so on that make up the mathematical literature. In our discussions so far, the term "modular" is applied to objects of this sort. But to make progress on questions related to the understanding of mathematics, we will have to make sense of some of the less tangible complements of a syntactic body of knowledge, namely, the concepts, methods, intuitions, and ideas that guide their use. It is far less clear how to speak rigorously of these. A *method,* for example, seems to be some sort of quasi-algorithmic entity that transforms one epistemic state to another, where the notion of an epistemic state may perhaps be represented as some sort of quasi-syntactic entity. A *concept*, like the group concept, may be viewed as a body of methods, clustered around a central definition or notion. (See [3, 4, 5] for some thoughts along these lines.)

I will not make progress on refining such talk here, but simply suggest that these more amorphous objects of knowledge can be modularly structured as well. Methods (or abilities or capacities) seem to be compositional: we can explain the ability to solve a problem in group theory in terms of the ability to invoke and apply relevant theorems, which in turn, may invoke the ability to construct particular instances of groups, to which the theorems are applied. Insofar as methods are like algorithms, and algorithms are represented by code, some of the things we say about modular code may transfer to talk of methods. Notions of interface may help explain how appropriate methods are triggered and applied, and what ensures that the results are not sensitive to the implementation. You and I can both carry out algebraic calculations, and that may be sufficient for us to understand a particular proof, even though we carry out the calculations in different ways.

If it does make sense to talk about modularity of the more dynamic components of knowledge, one would expect modularity of method to track modularity of syntax. Insofar as definitions, theorems, questions, and so on are (part of) the data on which our methods operate, a modular structuring of methods will necessarily depend on a modular structuring of the data.

## 6.2 Philosophical applications

In this section, I describe some of the ways in which the study of modularity may interact with other lines of inquiry in the philosophy of mathematics.

**Representations.** In cognitive science, psychology, and education, it is often held that understanding and cognitive competence rely on having the right *representations*. The notion is also a term of art in philosophy, playing a role in the philosophy of Descartes and Kant, for example, and contemporary philosophy of mind [33].

In Section 5.2, we saw that Fermat's Little Theorem can be expressed in various ways. Given a prime, $p$, and an integer $a$ not divisible by $p$, the conclusion can be expressed in any of the following forms:

- There is an $m$ such that $a^{p-1} = mp + 1$.
- $a^{p-1} \equiv 1 \bmod p$.
- $a^{|\mathbb{Z}_p^*|} = 1$ for any $a \in \mathbb{Z}_p^*$.

(In the last expression, $\mathbb{Z}_p^*$ is the multiplicative group of nonzero residues modulo $p$.) It seems that the best way to make sense of the differences between these representations is to consider them against a backdrop of a modular structuring of knowledge, where components of that body of knowledge interact with the representations in determinate ways, with suitable interfaces to mediate the interactions. A representation is useless unless one knows what to do with it; to paraphrase Kant, representations without interfaces are blind.

**Abstraction.** Understanding mathematical resources via modularity can help us understand the nature of abstraction, since specifying an interface is a way of characterizing an object in terms of its essential properties rather than its representation.

**Naturality.** Tappenden [40] has suggested that mathematical definitions seem to denote *bona fide* metaphysical entities rather than artificial or gerrymandered concepts when those definitions prove to play a fruitful or even critical role in our theorizing. The perspective offered here can provide an explanation of how they come to do so, namely, by contributing to modules and interfaces that have the desired effects.

**Generality.** The notion of modularity helps solve another puzzle. The virtue of introducing axiomatic and algebraic abstractions is usually attributed to their generality: for example, Dedekind's notion of an ideal, mentioned briefly in Section 5.3, has widespread uses in number theory, algebraic geometry, and functional analysis. But every such abstraction has to have an initial application, and unless that initial application yields an immediate payoff, it is hard to see how the abstraction can get off the ground. It is sometimes the case

that algebraic abstractions are introduced to unify existing theories and results, abstracting their common features. But not always: Dedekind introduced his theory of ideals to improve on Kummer's theory of ideal divisors, and was clearly pleased with the results, even before there were additional applications on the horizon. Indeed, it is often the case that algebraic and axiomatic abstraction provide a useful means of simplifying and clarifying a single proof or theoretical development. But that raises the question: is it just a coincidence that the kinds of abstractions that make proofs and theories more understandable often give rise to components that are reusable and more generally applicable?

Modularity explains the phenomenon by attributing both the improved understandability and the reusability to a common cause: the introduction of components with clear interfaces that make salient the essential data in a certain line of reasoning, and filter out extraneous information. Doing so makes a proof easier to understand, because there is less data to process and key relationships are easier to discern; but it also yields concepts and results that depend on fewer specific features of the context in which they are used, and hence are reusable and more general.

**Explanation.** There are various attempts, in the literature, to clarify what it means for a mathematical result to be *explanatory*. For example, Kitcher [25] takes explanation to be theoretical unification, while Steiner [38] expects an explanatory proof to make use of a "characteristic property" of an object mentioned in the theorem in a certain way. (These, and other approaches, are nicely surveyed in [27].) Because such analyses rely on structural notions of mathematical theories — the applicability of mathematical resources across different contexts, or the variability of proofs with certain parameters — the accounts may benefit from a clearer articulation of such structural notions.

**Purity.** Mathematicians sometimes express sentiments that promote certain kinds of purity of method, for example, sentiments to the effect that an elementary theorem should have an elementary proof, or that a geometric theorem should have a purely geometric proof. Andrew Arana and Michael Detlefsen have considered various notions of purity and the associated epistemic benefits (for example, in [2, 18]). The approach offered here may help clarify some of the claims. For example, the notion of purity explored in [18], *topical purity*, demands that a proof draw on only those axioms and definitions that are needed to determine the meaning of a theorem. Understanding the body of mathematics in the terms proposed here may help make sense of what determines that meaning; for example, notions of modularity can be used to screen out terms and facts that are deemed incidental to a particular presentation.

## 6.3 Summary

I have argued that it is possible to transfer, in a meaningful way, concepts and methods of analysis from the realm of software engineering to the philosophy

of mathematics. The transfer can be decomposed into two steps: insofar as formal languages used in interactive theorem proving are forms of computer code, notions from software engineering make sense when applied to formal definitions, theorems, proofs, theories; and insofar as the latter reflect important features of their informal mathematical counterparts, we can apply these notions to informal mathematics as well.

In particular, in computer science, modular structure is typically held to support understandability, reliability, the possibility of independent development, flexibility, and reuse. Transferring notions of modularity to mathematics provides insight as to how these benefits are achieved in that setting as well.

This perspective is largely orthogonal to traditional approaches to addressing ontological and epistemological questions. In particular, it seems equally compatible with realist and antirealist views of mathematics. But in addition to being independently valuable, a better understanding of what we value in mathematics, and why, can inform traditional lines of inquiry as well. For example, it can provide a more robust picture of how our mathematical language deals with mathematical objects, and what it is about such objects that gives them the air of reality.

There is still a lot of work to be done. One thing we can do is to continue analyzing the data — the historical and contemporary record of mathematical practice — in terms of the notions proposed here. At the same time, we need to develop better conceptual and logical models, with more precise ways of analyzing the structure of mathematical artifacts and assessing their epistemic value. Getting a grip on mathematical understanding will requires both philosophical analysis and careful attention to the mathematics itself, and so the two approaches should go hand in hand.

# References


[1] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition, 1996.

[2] Andrew Arana. Logical and semantic purity. *Protosociology*, 25:36–48, 2008.

[3] Jeremy Avigad. Mathematical method and proof. *Synthese*, 153(1):105–159, 2006.

[4] Jeremy Avigad. Understanding proofs. In Paolo Mancosu, editor, *The Philosophy of Mathematical Practice*, pages 317–353. Oxford University Press, Oxford, 2008.

[5] Jeremy Avigad. Understanding, formal verification, and the philosophy of mathematics. *Journal of the Indian Council of Philosophical Research*, 27:161–197, 2010.

[6] Jeremy Avigad and John Harrison. Formally verified mathematics. *Commun. ACM*, 57(4):66–75, April 2014.

[7] Jeremy Avigad and Rebecca Morris. Character and object. To appear in the *Review of Symbolic Logic*.

[8] Jeremy Avigad and Rebecca Morris. The concept of "character" in Dirichlet's theorem on primes in an arithmetic progression. *Archive for History of Exact Sciences*, 68(3):265–326, 2014.

[9] Carliss Y. Baldwin and Kim B. Clark. *Design Rules: The Power of Modularity Volume 1*. MIT Press, Cambridge, MA, USA, 1999.

[10] H. Clark Barrett and Robert Kurzban. Modularity in cognition: Framing the debate. *Psychological Revie*, 113:628–647, 2006.

[11] Manjul Bhargava. Higher composition laws. I. A new view on Gauss composition, and quadratic generalizations. *Ann. of Math.*, 159(1):217–250, 2004.

[12] Nicholas Bourbaki. The architecture of mathematics. *The American Mathematical Monthly*, 57(4):221–232, 1950. Translated from the French by Arnold Dresden. The original version appeared in F. Le Lionnais ed., *Les grands courants de la pensée mathématique*, Cahiers du Sud, 1948.

[13] J. Buchmann and U. Vollmer. *Binary Quadratic Forms: An Algorithmic Approach*. Springer, Berlin, 2007.

[14] W. Callebaut and D. Rasskin-Gutman. *Modularity: Understanding the Development and Evolution of Natural Complex Systems*. The Vienna series in theoretical biology. MIT Press, Cambridge, MA, USA, 2005.

[15] Peter Carruthers. *The Architecture of the Mind: Massive Modularity and the Flexibility of Thought*. Oxford University Press, Oxford, 2006.

[16] David A. Cox. *Primes of the Form $x^2 + ny^2$: Fermat, Class Field Theory, and Complex Multiplication*. Wiley, Hoboken, NJ, 2014.

[17] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover. In *25th International Conference on Automated Deduction (CADE-25)*., 2015.

[18] Michael Detlefsen and Andrew Arana. Purity of methods. *Philosophers' Imprint*, 11(2), 2011.

[19] Edsger W. Dijkstra. *Selected Writings on Computing: A personal Perspective*, chapter On the Role of Scientific Thought, pages 60–66. Springer New York, New York, NY, 1982.

[20] Harold M. Edwards. *Riemann's zeta function*. Dover Publications Inc., Mineola, NY, 2001. Reprint of the 1974 original [Academic Press, New York].

[21] Harold M. Edwards. *Essays in constructive mathematics*. Springer, New York, 2005.

[22] Jerry A. Fodor. *The Modularity of Mind*. MIT Press, Cambridge, MA, USA, 1983.

[23] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A machine-checked proof of the odd order theorem. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving*, volume 7998 of *Lecture Notes in Computer Science*, pages 163–179. Springer, 2013.

[24] Professor Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, New York, NY, USA, second edition, 2016.

[25] Philip Kitcher. Explanatary unification and the causal structure of the world. In Philip Kitcher and Wesley Salmon, editors, *Scientific Explanation*, pages 410–505. University of Minnesota Press, Minneapolis, 1989.

[26] Franz Lemmermeyer. *Reciprocity laws: from Euler to Eisenstein*. Springer Monographs in Mathematics. Springer-Verlag, Berlin, 2000.

[27] Paolo Mancosu. Mathematical explanation: why it matters. [28], pages 134–440.

[28] Paolo Mancosu, editor. *The Philosophy of Mathematical Practice*. Oxford University Press, Oxford, 2008.

[29] Kenneth Manders. Expressive means and mathematical understanding. mansuscript.

[30] Kenneth Manders. The Euclidean diagram. In Paolo Mancosu, editor, *The philosophy of mathematical practice*, pages 80–133. Oxford University Press, Oxford, 2008. MS first circulated in 1995.

[31] Gerald J. Myers. *Composite/structured design*. Van Nostrand Reinhold, New York, 1978.

[32] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, December 1972.

[33] David Pitt. Mental representation. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. 2012.

[34] Philip Robbins. Modularity of mind. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. 2009.

[35] John Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001.

[36] Herbert A. Simon. The architecture of complexity. *Proceedings of the American philosophical society*, 106(6):467–482, 1962.

[37] Herbert A. Simon. *Administrative Behavior, 4th Edition*. Free Press, New York, 1997. First edition, Macmillan, New York, 1947.

[38] Mark Steiner. Mathematical Explanation. *Philosophical Studies*, 34:133–151, 1978.

[39] Dirk J. Struik. *A Source Book in Mathematics, 1200-1800*. Source Books in the History of the Sciences. Harvard University Press, Cambridge, MA, 1969.

[40] James Tappenden. Mathematical concepts and definitions. In Mancosu [28], pages 256–275.

[41] Niklaus Wirth. Program development by stepwise refinement. *Commun. ACM*, 14(4):221–227, April 1971.