Modularity in Mathematics

Jeremy Avigad

Department of Philosophy and
Department of Mathematical Sciences
Carnegie Mellon University

May 2017

Overview

The term “modular” is a term of art in biology, computer science, business administration, architecture, neuroscience, cognitive science, philosophy of mind, . . .

Mathematical knowledge tends to be structured in modular ways.

The goal of this talk: to explore how, and why.

Overview

The thesis:

Mathematics (whatever that is) is modular (whatever that means), and that is good (in some sense).

It will take some effort to make this more substantial.

Overview

- Mathematics from a design standpoint
- The concept of modularity
- Modularity in mathematics
- Examples from number theory
- Conclusions

Modularity in mathematics

Initial questions:

- What is mathematics? (At least, what is it that can bear the predicate “modular”?)
- What are the goals of mathematics? (At least, what are the sorts of reasons that could explain the value of modularity?)

Methodological stance: focus on mathematical language, and on mathematical rules, norms, and values that can be described in linguistic terms.

Mathematical knowledge

We can distinguish between two sorts of objects of knowledge (or understanding):

- syntactic: definitions, theorems, proofs, theories, questions, conjectures, . . .
- more nebulous: methods, concepts, heuristics, intuitions, . . .

I will focus mostly on modularity of syntactic entities.

I suspect that if we can make sense of the second group, we will see that modularity of syntactic objects is just one aspect of modularity of method.

Mathematical goals

Mathematics aims to get at the truth.

But we can't check all natural numbers to see whether the Goldbach conjecture is true: we are finitary agents.

More than that: we are cognitively bounded. We can only do so much.

Mathematics aims to get us to the truth, *efficiently*.

For the time being, we'll have to rely on intuitive notions of complexity.

Philosophy of mathematics as a design science

Refined questions:

- What does it mean for definitions, theorems, proofs, theories, and so on to be modular?
- How does modularity help us do mathematics more efficiently?

Mathematical artifacts are designed to serve our purposes, and can do that poorly or well.

The philosophy of mathematics should tell us the principles that ensure that our mathematical artifacts serve our purposes well.

Modular systems

Herbert Simon, “The Architecture of Complexity,” 1962, spoke of “nearly decomposible” systems rather than modular ones.

Modularity has been studied with respect to:

- biology
- social organizations (like a business)
- hardware design
- software design
- architecture
- the mind

Modular systems

Roughly, a complex system is 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 *abstraction*: the function of the components can be described with respect to the behavior 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.

Modular systems

Modular systems can also be *hierarchical*.

This is not a necessary feature: one can have modular designs that are essentially flat.

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 by a hierarchical conception of its components.

Modular systems

A modular design is often claimed to bring certain benefits:

- *Comprehensibility:* makes it easier to understand, explain, and predict.
- *Independence:* allows the components of a system to be built and tested independently.
- *Reliability and robustness:* makes it easier to find and correct errors.
- *Flexibility:* makes it easier to change and adapt.
- *Reuse:* components that prove successful in one system can be used in others.

These are features we want our mathematics to have.

Modular systems

Design Rules: Volume 1. The Power of Modularity by Baldwin and Clark is about hardware design.

Modularity is characterized as “a particular design structure, in which parameters and tasks are independent 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.*

Strategy

We would like to apply such notions to mathematical artifacts.

We will look to interactive theorem proving, which provides semi-formal models of informal mathematics:

- The term “modular” is commonly used to describe software.
- Formal proof scripts are like computer code.
- Important features of informal mathematics are reflected in the formal proof systems.

This approach does not make strong presuppositions about formal vs. informal mathematics.

The formal models provide a useful means of analysis.

Modularity in computer science

Let's first consider modularity in computer science.

Fortran (1950's) and Basic (early 60's) result in "spaghetti code."

In the 1960's, methodologies developed to support subroutines, viewed as "modules."

Interactions with other pieces of code were mediated by the module's *interface*.

Modularity in computer science

In 1970: Niklaus Wirth (designer of *Pascal*), in "Program development by stepwise refinement" 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.*

Modularity in computer science

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.*

Modularity in computer science

In 1972, in “On the criteria to be used in decomposing systems into modules,” David Parnas wrote:

> *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. . .*

Modularity in computer science

He promoted "information hiding":

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

This is also known as "encapsulation."

Modularity in computer science

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.

Modularity in computer science

Summarizing:

- 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.

Characterizing modularity

Some examples of code (in a made-up language):

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

const pi : float := 3.1415

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

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

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

A definition associates an identifier (the *definiendum*) to an expression (the *definiens*).

Characterizing modularity

A bare-bones notion of syntactic dependence: one definition depends on another if the *definiens* of the first mentions the *definiundum* of the second.

Derivative dependences:

- syntactic well-formedness.
- semantics
- semantic properties

Characterizing modularity

What are the interfaces?

Various possibilities:

- the type
- the denotation (e.g. “what function it computes”)
- specific properties of the denotation

What is encapsulated is what is left out of the interface.

From programs to proofs

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 struture 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.

Robert Harper, *Practical Foundations for Programming Languages*

Interactive theorem proving

“Formal methods” refer to the use of formal, logic-based methods in computer science.

“Formal verification” can mean:

- verifying the correctness of hardware or software with respect to a specification
- verifying the correctnectness of a mathematical proof

“Interactive theorem proving” means working with a proof assistant to construct a formal axiomatic proof.

These are natural outgrowths of twentieth century foundational research.

Interactive theorem proving

Some theorems formalized to date:

- the prime number theorem
- the four-color theorem
- the Jordan curve theorem
- Gödel's first and second incompleteness theorems
- Dirichlet's theorem on primes in an arithmetic progression
- The Cartan fixed-point theorems
- The central limit theorem

There are good libraries for elementary number theory, real and complex analysis, point-set topology, measure-theoretic probability, abstract algebra, Galois theory, . . .

Interactive theorem proving

Some landmarks:

In 2012, Georges Gonthier and coworkers completed a verification of Feit-Thompson Odd Order Theorem in Coq.

In 2014, Thomas Hales and coworkers completed a verification of the Kepler conjecture (the *Flyspeck* project).

From programs to proofs

The analogy between programs and proofs is made explicit in interactive theorem proving, where proofs are "code."

The Feit-Thompson theorem was part of the *Mathematical Components* project:

> *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.*

From programs to proofs

From the final writeup:

. . . 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. . .

From programs to proofs

> 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 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. . .

From programs to proofs

The dialectic:

- The language of a proof assistant models informal mathematics.
- Text in such a language is a form of code.
- We know (more or less) how to talk about modularity in code.
- So it makes sense to talk about modularity in formal libraries.
- Insofar as these model informal mathematics, we can speak of modularity in mathematics.

Towards a formal model

In a conventional programming language, we have:

- data type specifications (`nat`, `float`, `point`, `float` $\rightarrow$ `float`)
- functions and data (`pi`, `circle_area`)

Proof assistants provide means to describe all the following:

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

This list is not meant to be exhaustive.

Towards a formal model

```
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,
take x y z,
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
```

Towards a formal model

An expression of one kind can depend on entities of other kinds:

- the definition of a mathematical object can depend on other objects and data types
- a proof can depend on data types, objects, propositions, and other proofs
- the definition of an object can depend on a proposition (e.g. `if even x then 0 else 1`)
- the definition of an object can depend on a proof (e.g. defining `gcd x y` as the greatest common divisor of `x` and `y`)

Towards a formal model

What are the interfaces?

- types
- algebraic structures (e.g. instantiating the reals as an ordered ring)
- theory files, namespaces, module systems, . . .

Measures of complexity

Modularity is supposed to make code easier to understand, easier to develop and 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 modular mathematical theories has the same benefits.

This presupposes some conception of what it means to be easier to understand, maintain, or extend a theory.

Measures of complexity

We would like to consider the benefits of modularity with respect to mathematical proofs.

Two possible measures:

- how hard it is to *find*, or *discover*, a mathematical proof
- how hard it is to read and understand a proof

Let's focus on the latter. Even that is no small task.

Measures of complexity

Reading a proof is a complex task: 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.

How does a modular structuring of the proof and the background theory help?

Measures of complexity

Modular structuring decreases the cognitive burden:

- Type interfaces makes it possible for us to infer the types of objects and expressions in front of us.
- Types and algebraic structures make it easier to apply theorems and constructions.
- Types, algebraic structures, and modular structuring of theories makes it easier to find and retrieve relevant facts from our background knowledge.
- Encapsulation keeps information overload at bay.

This is the beginning of an answer. We need more substantive theories of how this works.

Examples

I have made the case that modularity *should* be valued in mathematics.

In fact, modularity is everywhere:

- Books are divided into chapters.
- Proofs are broken down to lemmas.
- Subjects and bodies of knowledge are broken down into smaller disciplines.

Concrete examples will help us think about how the notions play out.

Congruence

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

Definition. If m is another integer, say x *is congruent to* y *modulo* m, written $x \equiv y \pmod{m}$, if $m \mid x - y$.

Let us consider a toy, but illustrative, example:

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

Congruence

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 doesn't scale.

More significantly, it breaks abstraction.

Congruence

Proposition. 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 1, we have $x = x \cdot 1$. For 2, if $y = xu$ and $z = yv$, then $z = x(uv)$. For 3, if $y = xu$ and $z = xv$, then $y + z = x(u + v)$. For 4, if $y = xu$, then $zy = x(zu)$. For 5, take $y = x$ and $z = 0$ in 3. □

This is the only place where we need to unfold the definition of $\mid$.

Congruence

Proposition.

1. $\equiv$ is an equivalence relation.
2. If $x \equiv y \pmod{m}$, then $x + z \equiv y + z \pmod{m}$
3. If $x_1 \equiv y_1 \pmod{m}$ and $x_2 \equiv y_2 \pmod{m}$ then $x_1 + x_2 \equiv y_1 + y_2 \pmod{m}$.
4. If $x \equiv y \pmod{m}$, then $xz \equiv yz \pmod{m}$.
5. If $x_1 \equiv y_1 \pmod{m}$ and $x_2 \equiv y_2 \pmod{m}$ then $x_1 x_2 \equiv y_1 y_2 \pmod{m}$.
6. If $x \equiv y \pmod{m}$, then $x^n \equiv y^n \pmod{m}$ for every natural number n.

It follows that if $p(x)$ is any polynomial with integer coefficients and $x \equiv y \pmod{m}$, then $p(x) \equiv p(y) \pmod{m}$.

Congruence

In the refactored version:

- the existential quantifier in “divides” encapsulates data.
- the proofs about congruence respect that interface.

Benefits of the refactoring:

- The proof is easier to understand.
- The properties of divisibility and congruence are reusable.
- The result is more general.

Think about what is encapsulated with $\lim_{x \to a} f(x) = b$.

Fermat's Little Theorem

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

Euler published a proof in 1761.

First, he shows that for any a and $p \nmid a$, there is a least $\lambda > 0$ such that $a^{\lambda} \equiv 1 \pmod{p}$.

He also showed that the residues of $\{1, a, a^2, \ldots, a^{\lambda-1}\}$ are distinct.

We would say that the nonzero residues modulo p form a group under multiplication modulo p.

Fermat's Little Theorem

Theorem. 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.

Proof. Let a 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 λ 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 λ and, reducing them in the usual way, they all become smaller than p and are all different from each other. As λ 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

Fermat's Little Theorem

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^μ; 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^μ 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 λ numbers not belonging to the residues so long as $\lambda < p - 1$.

Fermat's Little Theorem

The rest of the proof runs three times as long as the excerpt.

What does a modern proof look like?

Fermat's Little Theorem

Let G be the group of nonzero residues modulo p.

Let $H = \{1, a, a^2, a^3, \dots, a^{\lambda-1}\}$ be the subgroup generated by a.

If k is any element of G, let Hk denote the *coset* $\{hk \mid h \in H\}$.

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

Proof. If $h_1 k = h_2 r$, then $k = h_1^{-1} h_2 r \in Hr$. □

Proposition. For any k in G, $|Hk| = |H|$.

Proof. $h \mapsto hk$ is a bijection from H to Hk; $h_1 k = h_2 k$, then $h_1 = h_2$. □

Fermat's Little Theorem

Proposition. $|H|$ divides $|G|$.

Let $g_1 = 1$. If $H = Hg_1 \neq G$, pick an element g_2 in G but not Hg_1. If $Hg_1 \cup Hg_2 \neq G$, pick an element g_3 in G but not Hg_1 or Hg_2, and so on. Since G is finite, eventually we obtain

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

The sets $Hg_1, Hg_2, \ldots, Hg_n$ are disjoint and each has cardinality $|H|$, so $|G| = |H| \cdot n$. □

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

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

Fermat's Little Theorem

Why is Euler's proof so much longer?

Part of it stylistic: he makes little effort to be concise.

We also help ourselves to set-theoretic terminology, with the coset definition and the notion of a bijection.

But the main thing is the Euler doesn't use our abstraction: once we know that the residues form a group, we never have unfold the "definition" of multiplication.

Fermat's Little Theorem

The proof, in fact, establishes Lagrange's theorem: if H is a subgroup of a finite group G, $|H|$ divides $|G|$.

As a corollary, we get Euler's theorem: if a and m are coprime, $a^{\varphi(m)} \equiv 1 \pmod{m}$.

Benefits of the refactored version:

- the proof is easier to check and understand
- we get much stronger generalizations

Other examples

Nineteenth century number theory provides nice examples:

- problems are usually easy to state
- they often require serious machinery
- results are reworked and generalized dramatically

Good case studies:

- quadratic reciprocity
- binary quadratic forms
- Dirichlet's theorem
- the prime number theorem

Conclusions

It is generally undertood that modularity brings benefits to software design:

- understandability
- reliability and robustness
- independence
- flexibility and adaptability
- generalizability and reuse

Our goal has been to transfer that to mathematics.

Conclusions

The dialectic:

- Formal mathematics, as developed in interactive theorem provers, is like computer code.
- Relevant features of informal mathematics are captured in these formal languages.

There is a lot we need to do to flesh out the model:

- Pay careful attention to the data — the historical and contemporary record of mathematics — and study it in terms of these notions.
- Develop better conceptual and logical models, with more precise ways of analyzing the structure of mathematical artifacts and assessing their epistemic value.

Conclusions

This type of analysis is likely to inform other topics of interest in the philosophy of mathematics (and philosophy in general):

- representation
- abstraction
- naturality
- generality
- explanation
- purity