Universal Properties, Abstraction Barriers, Isomorphism Invariance, and Homotopy Type Theory

Jason Gross

Course Information
Course Number: TBD
Credit Hours: 3-0-9
Meeting Times: TBD
Classroom Location: TBD
Website URL: TBD

Instructor Information
Jason Gross
(he/him/his)
Office: 32-G888
Office Hours: TBD
Email Address: jgross@mit.edu

Course Description

Mathematicians interact with many mathematical objects over their courses of study. We learn in grade school about natural numbers, integers, rational numbers, and later real numbers; we talk about functions in high school. We perform geometric constructions. In undergrad mathematics, we might interact with matrices, linear transformations, vectors, vector spaces, sets, topological spaces, groups, rings, fields, modules, and more. Finally, we read and write proofs, which, depending on whom you ask, might or might not be mathematical objects in and of themselves.

Through osmosis, you may have picked up a vague notion of what a mathematical object is allowed to be, and what it means to have one. Some basic questions, though, have likely been swept under the rug. Is a red "3" the same as a blue "3"? If you take the formal set-theoretic definition of natural numbers where $0 = \emptyset$, $1 = \{0\}$, $2 = \{0, 1\}$, ..., is this 0 "the same as" the 0 you'd get if you started off with $0 = \{\emptyset\}$, $1 = \{0\}$, $2 = \{0, 1\}$, ...?

In this class, I will aim to convey a particular mathematical aesthetic, a lens through which you can look at not just the world of math, but the world at large. The underlying, unifying thread of this class will be the question "What is a mathematical object?", or, said another way, "What is the structure of the rules by which we play this game of math?"

We'll be covering some basic category theory: a universal property is a way of pinning down a mathematical object in a way that is unique up to isomorphism

(and the isomorphism itself is in fact unique). We'll look at the relationship between abstraction barriers and APIs in computer science and mathematical objects. Finally, we'll dive into some basic Homotopy Type Theory, an exciting new field of math which provides an alternative to set theory as mathematical foundations, and allows stating, proving, and using the Univalence Axiom, that isomorphic objects can be considered equal.

Prerequisites/Corequisites

Intended Learning Outcomes

By the end of this course, students will be able to...

- see the world through the lens of objects being defined only and uniquely by their relationships with other objects; (Note: I still don't know how to asses this one directly, but I'm hopeful that my unit-level assessment below implicitly requires enough of this lens that I can assess it (maybe? hopefully?))
- know and understand the basic category-theoretic definitions—that is, students should feel comfortable using fluidly definitions including those of category, functor, natural transformation, isomorphism, initial and terminal objects, and adjoint functors, and will be able to justify why each component of the definition is useful or necessary, and why no additional properties are needed;
- know definitions and examples of isomorphisms, universal properties, and what it means to identify something uniquely up to unique isomorphism; know also how to apply these definitions to new problems;
- approach problems from a morphism-centric or relation-centric view rather than an object-centric one;
- identify when two theorems, conjectures, proofs, or problems in different areas of math are in fact the same;
- describe their frames, lenses, or aesthetics for approaching problems, as well as be able to discover others' lenses with enough clarity that they can describe those lenses even if they'd never been exposed to that particular lens before;
- feel comfortable encountering, making sense of, and making use of category-theoretic language and framing;
- turn standard mathematics definitions into definitions phrased in terms of universal properties

Course Structure

The content of this course is organized into the four major units named in its title:

1. Universal Properties
2. Abstraction Barriers
3. Homotopy Type Theory
4. Isomorphism Invariance

Briefly, the unit on universal properties aims to give students the tools to formally talk about mathematical objects, relationships between them, and how and what it means to characterize a kind of mathematical object. In the unit on abstraction barriers, we will take this formal grounding to the world of computer science and learn how to apply the lens of the first unit to computational objects and computer systems. In the third unit, we will return to the foundation of mathematics, and look at how the foundations of mathematics can involve characterizations based on universal properties and even computation; the primary foundations we'll investigate are set theory, presented through the lens of the *Elementary Theory of the Category of Sets*, and Homotopy Type Theory. The final unit aims to make an explicit and formal rendition of an underlying theme or assumption of the lens used throughout this class: that we only care about objects up to isomorphism.

Unit: Abstraction Barriers: Extended Description

Having covered basic category theory, we will now take this knowledge and apply it to computation. The essential idea behind abstraction is that, having built a compound object out of multiple components, we can throw away our knowledge of how the compound object was *constructed* without losing anything important. We will explore this idea with a variety of examples which may include:

- Building integers in set theory
- Building rational numbers out of integers
- Building complex numbers from real numbers
- Building fixed-size integers out of bits
- Building unbounded integers out of fixed-size integers (typically 32 or 64 bits on computers)
- Building integers out of pure functions

- Building lists out of pointers
- Building binary trees out of pointers
- Building lists out of cons cells in Lisp
- Building lists out of pure functions
- Building finite functions from lists
- Building source code from strings
- Building source code from trees
- Building theorem statements and formal proofs from integers (Gödel codes)
- Building theorem statements and formal proofs from trees

We will discuss what it means to respect abstraction barriers and what it means to pierce them, as well as reasons for doing each, in each example.

After achieving an solid foundation of what abstractions are, we will revisit the examples to characterize the objects we are constructing with the language of universal properties. The key question here is "how do we describe the object we are constructing in a way that identifies it uniquely up to unique isomorphism?" In answering this question, we will also investigate related questions such as:

- Does what description we get depend on what category we're working in?
- What actually are the objects and what are the morphisms?

Having a formal grounding for describing abstractions with universal properties, we will turn towards larger examples in computer science. In this part of the unit, we will consider especially when a provided interface seems to mismatch the intended object being described, and students may be asked to investigate the costs of mismatch.

Finally, students will be asked to turn this reasoning on the unit topic itself: How can "abstraction" or "abstraction barriers" be identified abstractly? What would a characterization via a universal property look like? What sort of object is an "abstraction barrier"? What other objects live in the same category, and what might the morphisms look like?

Unit: Abstraction Barriers: Intended Learning Outcomes

At the end of this unit, students will be able to:

- Define "abstraction" and "abstraction barrier" in their own words
- List examples of abstraction in both mathematics and computer science
- Identify where abstraction is going on, in code bases and mathematical texts
- Identify when and how an API (application programming interface) mismatches the intended object being defined
- Refactor code and/or proofs which pierce an abstraction barrier in a way such that the abstraction barrier is respected
- Describe APIs and mathematical objects in terms of universal properties
- Identify when two objects, abstractions, proofs, or theorems are the same
- Think creatively and critically about picking good abstractions for a problem, algorithm, or proof

Unit: Abstraction Barriers: Summative Assessment

You may pick one of the following two projects for this unit. You may work in groups or individually; each project may either be presented to the class, or written up and submitted to the class for comments, or both. (Each student should participate substantially in at least one of a write-up or a presentation for whatever group they are in; this project will serve as a measuring stick for students' abilities to apply the concepts in this class.)

In either case, you will be expected to additionally write up a short document on your process: how you approached the assignment, what you discovered and learnt, what was easy and what was challenging, any advice you wish you had been given before starting the assignment, etc. The purpose of this document is both for me to understand how you're engaging with the material and where you are in the learning process, and for you to reflect on your learning and solidify any insights you might be able to draw from the process. Students who prefer oral assessments may instead schedule a short (15–45 minute long) interview with me where we discuss the aforementioned reflection.

Note that the first of these projects is tailored towards students who want to lean on previous exposure to computer science, while the second is tailored towards students who would prefer to lean on their previous exposure to abstract mathematics. I encourage you to pick whichever option appeals to you most; feel free to reach out to me by email, after class, or in office hours to discuss your choice if you are struggling or want guidance or support.

The two choices:

1. Find an open-source software project on GitHub (or similar) and identify where this codebase uses abstractions. Identify locations where some code in this codebase could be refactored to better respect an existing abstraction, or else to use an abstraction not previously present. (Groups should identify at least as many distinct abstractions being violated as there are students in the group, but there is flexibility in this criterion if some abstraction is violated in a large number of distinct ways or locations.) Estimate the costs (in programmer time, in development cost, in likelihood of having bugs, in performance, and/or in other metrics) of not using abstraction in these locations, and estimate as well the costs of using the abstraction.

   You are encouraged to find a message or insight and structuring your presentation or write-up around supporting this claim using your findings as an example. Example messages include: "abstraction is always good," "abstraction, when used correctly, is beneficial," "over-abstraction has severe costs," "abstraction is always bad," "(*insert a kind of abstraction violation here*) is pervasive and costly." I encourage finding such a claim both because I believe that it lends coherent purpose and structure to the presentation or write-up; and because I expect that the process of looking for and defending such a claim will allow you to extract more insight from this project.

2. Create a coherent and crisp way of defining "abstraction" and "abstraction barriers" abstractly through the lens of universal properties. You should address at least the following questions (or else justify why some of these questions are not relevant):

   - Is the characterization general enough? To what extent do the examples we've covered in class fit into this notion of abstraction? For examples that don't fit, to what extent is the lack of fit an indication that the example is somehow a false example or non-central, and to what extent does it point at a deficiency in the characterization?
   - Is the characterization specific enough? Does it capture anything that we don't want to call abstraction? Does it capture anything that seems surprising but fitting?
   - What new insight or clarity does this characterization bring to the concept of "abstraction" and "abstraction barriers" as a whole? What might we (me, you, or another class member) have missed about abstraction that becomes clear when using this characterization?
   - To what extent is the characterization precise and formal? Is it always clear, based on this characterization whether something is or is not an abstraction (or abstraction barrier)?

- If abstractions (and/or abstraction barriers) are objects in some category, what is the category (what are the morphisms between abstractions / abstraction barriers)? If "abstraction" (and/or "abstraction barrier") is itself a single object in some category, what is that category (what are the other objects, and what are the morphisms)? Note that abstraction might be both an object in some category while at the same time abstractions themselves are objects in some other category; an example of this occurring is with the concept "category": there is a category of (small) categories, which is itself an object in the category of (larger) categories.
- Is your characterization of abstraction itself an example of abstraction, according to itself?

Note that if you choose this option and choose to work in a group, I still expect each individual student in the group to come up with their own ideas (even if the ideas end up overlapping or being inspired by one another). I therefore encourage you to grapple with these questions and come up with your own ideas before discussing them with your group members.