A Homotopy Theory of Object-Oriented Programming

Author: Joseph B. Axenroth

Abstract

In object-oriented programming, the notion of a class (or typed data structure with bound behaviors) can be defined as a topological space and interpreted as a presheaf. We can then form higher categories for these classes and develop a homotopy theory to model their interactions. Interfaces provide programmatic invariances that may be treated at homotopy equivalences between implementing classes. One finds that object-oriented programming is a manifestation of a homotopy theory of higher categories.

Section 1: Introduction

In object-oriented programming (OOP), we have the notion of a class. Or, in some other respect, the notion of a typed data structure possibly associated with bound behaviors. The typed structure consists of a collection of states X. There is always at least one state, the identity state 1_X. In addition, the typed structure may include some number of properties, thus establishing a complexity of other states $x_i \in X$.

Although not required, it is useful to bind and associate behaviors with the typed structure. In OOP, these are referred to as methods; we'll denote the collection of methods as M, with some particular method as m_i, dropping the subscript as convenient. They allow one to change the state of some instance c_i of a class (or typed structure) C. For simplicity, assume idempotency for methods, so each m only affects internal state with no side effects. One can then interpret a method m_i as a means of providing a 'path' from one state x_i to another x_j. We'll denote a path as p, with p_i as some particular path.

Yet the story is a bit more complex than this. A collection of states X may be valid for use with the method, while a disjoint collection of states Y may not be valid. For each x_i in the collection of valid states, the method maps to a (possibly empty) collection X' of states. Note that X may be equivalent to X', with equivalency here defined by membership. The collection

shall be indexed by the input state x_i in conjunction with any potential collection of input parameters T to the method.

So a method maps states in a collection X to a collection of families of states, $\sum X_i \subseteq X$. While technically possible, these families need not be disjoint. It may even be possible that all the X_i are equivalent, with equivalency again defined by membership.

For any of the families X_i, there is a collection of paths P_i from the originating state x_i to one of the ending states x_j, with $x_i, x_j \in X_i$. All of the paths $p \in P_i$ are homotopically equivalent to one another. In turn, this homotopy equivalence is itself homotopy equivalent to the induced homotopy equivalences of all the other families X_j.

All of this only works, however, if we have a suitable model for the classes, that allows us to properly define the homotopies. We can define a class in OOP by using topological spaces. If we interpret a class (as a type) as a (topological) space, then the state of a class (type) corresponds to a point in the space. In fact, this is one of the general notions of homotopy type theory (HoTT) [HoTT13]. Except with HoTT, spaces are regarded homotopically and strictly not topologically; there is no notion of open sets, yet we will need the notion for our theory.

Going further, we can interpret classes in OOP using presheaves. The index category has as objects the methods. Arrows can be defined such that a first method m_i, as the domain object, calls a second method m_j, as the codomain.

The presheaf maps to a set of collections of states X for which the class may exist in during a call to the method m_i. This includes a collection of states X_i for the start of the method and a collection of states X_j at the end of the call. Some state $x_i \in X_i$ has a path p to to some $x_j \in X_j$. Note that all the collections of states $X_i, X_j \subseteq X$ are assumed to be open sets.

We are only taken so far with this setup, however. Programmers do not use classes by themselves; classes are used with other classes. Types use other types. With some languages, classes may serve as a base class, with functionality (via properties and/or behavior) extended by other classes (establishing an inheritance hierarchy). Groups of classes may implement a common interface. These last two aspects correspond to the notion of polymorphism.

Let us consider an interface; for now we will ignore inheritance. Interfaces can declare, but not implement, methods. Classes then implement the methods for that interface. Any such implementing class may be used anywhere that requires that interface, and so interfaces provide a programmatic invariance between classes. They provide a way for classes to be polymorphic. In essence, they provide a way for classes to be treated as equivalent.

We can go further. We are able to say that interfaces provide a homotopical equivalence between implementing classes. Yet the homotopy equivalence via some interface would need to be properly composed as essentially a family of homotopy equivalences corresponding to the methods defined by the interface (and implemented by the classes).

In addition, there are other homotopy equivalences to consider. A method may have some number of parameters $t_0, ..., t_n$, $0 \leq n$. The parameter t_i may be of some interface type. Surprisingly, this does not add much additional complexity to the model of homotopy equivalence families for methods already described.

We find that a higher structure of homotopy equivalences come into play. Homotopy equivalences of homotopy equivalences of homotopy equivalences. At this stage of the story, we find that OOP is simply a manifestation of a homotopy theory of higher categories.

Section 2: Spatial Interpretations

In this section, we establish classes as spaces and develop their associated topology. From there we will be able to interpret classes using presheaves and examine the anatomy of a method.

Section 2.1: Establishing a Class as a Topological Space

Suppose we have a class C in object-oriented programming. As stated in the introduction, classes will have a collection of states X. If no other states exist, there will always be the identity state 1_X. In this case, X will be a singleton set, $\{ 1_X \}$. Otherwise X will contain additional states $x_0, ..., x_n$, $0 \leq n$. In object-oriented programming, classes are regarded as types. In HoTT [HoTT13], types are regarded as spaces. Therefore, we can regard a particular state $x_i \in X$ as a point in the corresponding space, with X as the space.

In order to define a topology for this space, we must first consider what will be the open subsets. When considering when a subcollection of states can be meaningfully differentiated in a class, one option is the collection of input (output) states that may be used with the various methods in the class. Denote as X_i a subcollection of states used as input to some method m, and X_j as the subcollection of states serving as output to that method. All the input/output subcollections X_k will serve as our open subsets.

For any arbitrary number of subcollections of X, we can define a method m that can take a union of these subcollections as either the input or output states. To do so may be dependent on the definition and structure of the class. Similarly, we can do the same for an intersection of

arbitrary number of subcollections of X. On such cases, this typically correlates with refactoring out a common method from multiple methods.

The null set $\varnothing$ will be part of the topology, corresponding to an empty set of states. Clearly the only case there is an empty set of states corresponds to a null object for a class, established before some object is instantiated via the class's constructor. Once instantiated, we are provided with the identity state at a minimum.

The collection of states X for the class will also be part of the topology. One can define a no-op method, or even a method that does perform some calculation, for which every state is valid, for either the input or output.

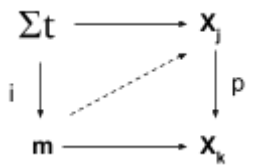
Section 2.2: Interpreting a Class as a Presheaf

We have established that a class can be regarded as a topological space, but in addition we can also interpret a class as a presheaf. The limiting aspect of a class as simply a topological space is that it does not quite capture the notion of methods, or the behaviors associated with the class. Define a class as a presheaf $C : M^{op} \to \Sigma X$. As indicated in the introduction, M denotes the collection of methods for a class, so we must define a category for this collection. The set ΣX corresponds to all the subcollections of states associated with the methods via the topology.

Section 2.2.1: Anatomy of a Method

There is a potential collection of parameters T for each method $m \in M$ in a class C. Assume $t \in T$ is of some interface type. It is possible that all $t \in T$ are of the same interface type, or that none share a type. Let us denote by Σt_i as the combinatorial product of the parameters for some method m_i. The subscript i can be dropped as needed when no confusion arises.

If we have X_j as a collection of states as input to a method m, and X_k as the collection of output states, then we have the following commutative diagram:



This diagram tells the story of a method in a class. The left arrow assigns a collection of parameters to the method m. The right arrow corresponds to some collection of input states

transitioning to a collection of output states. The top arrow pairs the input parameters to the input states; while injective, this arrow is clearly not surjective. And the bottom arrow establishes that a method can be used to determine a collection of output states.

In addition, a collection of input parameters to a method used to determine a collection of output states is equivalent to a collection of input parameters applied to input states causing a determination of a collection of output states. Notice the arrow providing a lifting. This lifting, unique to the method m, indicates the method can be applied to a collection of input states. Once applied, we are provided the output states.

We can interpret a method m, taking some state x_j to a state x_k, with a path p_j transitioning from x_j to x_k. Typically, a method will allow transitioning from any of several states to corresponding other states; as a result, we are provided with a collection of paths p. With respect to m, all of the paths p are equivalent with one another, up to homotopy. Anatomically speaking, a method m is effectively a homotopically equivalent family of paths.

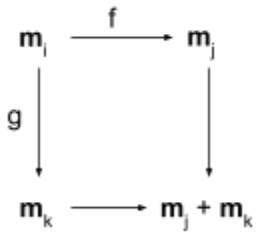
Section 2.2.2: Establishing the Collection M of Methods as a Weak Category

Suppose we have a collection of methods M for some class, with elements m in the collection. It is straightforward to define a weak category on this collection. The objects will be the elements. An arrow f maps some method m_i (the source of f) to a method m_j (the target of f). One potential way m_i can map to m_j is for m_i to use m_j; in other words, $f : m_i \to m_j$ can be interpreted as m_i calling m_j. This affects how m_i transitions from one state to another in a particular way. As part of that transition, m_j is called, which itself transitions a state from one to another.

We can form the composition of arrows as follows. Suppose we have arrows $f : m_i \to m_j$ and $g : m_j \to m_k$, so that $gf : m_i \to m_k$. What results is that the method m_i calls m_j which itself calls m_k. There are no identity arrows for objects (methods), which is what causes the category to be weakened. The reason why there are no identity arrows is that methods typically do not call themselves, except under recursion; clearly not every method is recursive.

For any method m as an object of the category M, there is a left and right unit arrow. In an abuse of notation, we denote the units as 1_m in both cases (the context of its use will be obvious if we are using the right or left unit). The left unit is a (possibly implicit) no-op method within the class, and the right unit is a wrapper method, whereby it is no-op up to calling the codomain method m_j. Assume the arrow f from the above paragraph with the left unit arrow 1_m, then we have $1_m \circ f$. The arrow f results in a change of state, with the resulting state unchanged by 1_m. Similarly for the right unit, with $f \circ 1_m$.

It is entirely possible, and even likely, that one method m_i may call multiple other methods. Assume m_i calls two methods m_j and m_k. Then during the course of state transitions for m_i the state will be transitioned intermediately twice, once for m_j and again for m_k. We then have $f : m_i \rightarrow m_j$ and $g : m_i \rightarrow m_k$, resulting in the following pushout square:



Similar (non-square) diagrams can be constructed when a method m_i uses (calls) three or more other methods, resulting in the coproduct of several methods $m \in M$. Note that recursive calls are a potential in the coproduct chain, whereby some method m_r calls itself with $r : m_r \rightarrow m_r$ (note this should not be confused with an identity morphism). Yet this should not complicate the scenario any further. Considerations for how this coproduct should be interpreted will be deferred to a later section.

Section 2.3: Interactions between Classes

We have established that the collection M of methods for some class C form a weak category. However, as currently defined, arrows portraying the interactions of methods are only established within a single class. Yet programmers will implement methods which interact with methods from other classes. So we need a means to model this with our theory.

Suppose we have some method $m_1 \in M_1$ as part of some class C_1, and similarly $m_2 \in M_2$ for another class C_2, with M_1 and M_2 the collections of methods for their respective classes. Also assume that m_1 calls m_2. We need some arrow $u : m_1 \rightarrow m_2$ formalizing m_1 of C_1 calling m_2 of C_2. The call from a method in one class to a method in a second class may actually be interpreted as a functor between the weak categories M_1 and M_2, or $F : M_1 \rightarrow M_2$.

Section 3: Interfaces as Homotopy Equivalences

In this section we will construct the tools necessary to take interfaces in object-oriented programming as programming invariants so we may treat them as homotopy equivalences, and explore what repercussions result from that.

[**References and Influential Works**]

[JLK55] *General Topology*. John L. Kelley. 1955. Van Nostrand Reinhold Company.

[SW70] *General Topology.* Stephen Willard. 1970. Addison-Wesley Publishing Company.

[BM75] *Introduction to Topology (3rd edition)*. Bert Mendelson. 1975. Allyn and Bacon, Inc.

[PTJ77] *Topos Theory*. P. T. Johnstone. 1977. Academic Press, Inc.

[CEIT78] *Counterexamples in Topology (2nd Edition)*. Lynn Arthur Steen and J. Arthur Seebach, Jr. 1978. Springer-Verlag.

[CRFM80] *Algebraic Topology*. C.R.F. Maunder. 1980. Cambridge University Press.

[RG84] *Topoi: The Categorial Analysis of Logic (2nd edition)*. Robert Goldblatt. 1984. Elsevier Science Publishers B. V.

[BCP91] *Basic Category Theory for Computer Scientists.* Benjamin C. Pierce. 1991. The MIT Press.

[MIT94] *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*. Carl A. Gunter and John C. Mitchell (editors). 1994. The MIT Press.

[GoF95] *Design Patterns: Elements of Reusable Object-Oriented Software*. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. 1995. Addison-Wesley.

[RBOD97] *The Algebra of Programming.* Richard Bird and Oege De Moor. 1997. Prentice Hall Europe.

[SML98] *Categories for the Working Mathematician (2nd edition)*. Saunders Mac Lane. 1998. Springer Science+Business Media, LLC.

[OXFD98] *Twenty-Five Years of Constructive Type Theory*. Giovanni Sambin and Jan Smith (editors). 1998. Oxford University Press.

[BJ99] *Categorical Logic and Type Theory*. Bart Jacobs. 1999. Elsevier Science B. V.

[JRM2k] *Topology (2nd Edition)*. James R. Munkres. 2000. Pearson Education, Inc.

[BCP02] *Types and Programming Languages*. Benjamin C. Pierce. 2002. The MIT Press.

[MF03] *Patterns of Enterprise Application Architecture*. Martin Fowler. 2003. Pearson Education, Inc.

[DDD04] *Domain Driven Design: Tackling Complexity in the Heart of Software*. Eric Evans. 2004. Pearson Education, Inc.

[JL09] *Higher Topos Theory*. Jacob Lurie. 2009. Princeton University Press.

[SA10] *Category Theory (2nd edition)*. Steve Awodey. 2010. Oxford University Press.

[CS12] *Homotopy Theory of Higher Categories: From Segal Categories to n-Categories and Beyond*. Carlos Simpson. 2012. Cambridge University Press.

[HoTT13] *Homotopy Type Theory*. Univalent Foundations of Mathematics. 2013. The Univalent Foundations Program.

[RNHG14] *Type Theory and Formal Proof: An Introduction*. Rob Nederpelt and Herman Geuvers. 2014. Cambridge University Press.

[RCM17] *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Robert C. Martin. 2017. Pearson Education, Inc.

[JB18] *The Homotopy Theory of (∞,1)-Categories*. Julia E. Bergner. 2018. Cambridge University Press.

[DC19] *Higher Categories and Homotopical Algebra*. Denis-Charles Cisinski. 2019. Cambridge University Press.