Formalizing Operads for a Categorical Framework of DSL Composition

Zachary Flores Angelo Taranto Yakir Forman
Two Six Technologies
zachary.flores@twosixtech.com angelo.taranto@twosixtech.com yakir.forman@twosixtech.com
Eric Bond
University of Michigan
bonderic@umich.edu

How do you build strong type safety into a programming language? One answer to this problem is to provide a formalization for, if it exists, the denotational semantics of the programming language. Achieving such a formalization provides a high standard for ensuring the programming language is correct-by-construction. In our paper, we discuss steps toward building the foundation for the denotational semantics of a meta-language for composition of high-level abstractions of domain-specific languages using operads. The category of operads has properties that allow for the smooth composition of objects, allowing us to easily build larger structures from basic pieces. In particular, these properties provide an excellent ground to model our meta-language and its need to compose high-level abstractions of domain-specific languages. To take the first step towards this formalization, we formalize operads, as they are the basic pieces future work relies on. Throughout this paper, we discuss our formalization of the definition of an operad in the proof assistant Coq and an important instantiation of our definition in Coq which provides a concrete formalization of an example of an operad. As such, this work in Coq provides a formal mathematical basis for our meta-language development. Our work also provides, to our knowledge, the first known formalization of operads within a proof assistant that has significant automation, as well as a model of operads that does not rely on Homotopy Type Theory.

Funding: Supported by DARPA V-SPELLS HR001120S0058.

1 Introduction

The DARPA V-SPELLS (Verified Security and Performance Enhancement of Large Legacy Software) program aims to create developer-accessible capability for piece-by-piece enhancement of software components for large legacy codebases with new verified code that is safely composable with the rest of the system.

In our approach with the Johns Hopkins Applied Physics Laboratory to solving the problems posed by V-SPELLS, our tool in development, called LUMOS, begins by applying methods from static analysis, natural language processing, and dynamic analysis to the legacy source code in order to generate high-level abstractions of these **domain specific languages** (DSLs) that we call **domain-specific semantic models** (DSSMs) from the DSLs that comprise the source code. The DSSMs will be generated in a

Submitted to:
MSFP 2024

language we refer to as the **meta-DSL**, and in order to provide the patches to the legacy code requested in V-SPELLS, these DSSMs will have to be composed in very specific ways. In order to ensure correctness of composition, as is required in V-SPELLS, we are providing verification of composability via an algebraic framework using several ideas from category theory, including the key structure to our modeling: **operads**. Operads, which are also known as **symmetric multicategories**, have begun to play an increasingly important role within applied mathematics (see [7, 4, 1, 2, 5]), and we find they provide an excellent mathematical model for our verification needs on V-SPELLS. To be more precise about our formal modeling, when a DSSM is written in the meta-DSL, we will use an operad to model the DSSM in the meta-DSL, and composition of DSSMs within the meta-DSL will be modeled via a gluing operation in the category of operads. Mathematically, operads provide will provide a formal model for the syntax of a key portion of the language of the meta-DSL, and we briefly discuss how this operad-based syntactical model can further be used to provide a mathematical model for the denotational semantics of the meta-DSL. However, we maintain the focus of this paper is to discuss our formalization of operads in the proof assistant Coq. Our repository for our formalization of operads can be found in [3].

2 Motivation

2.1 Role of Formalization in LUMOS

The initial need for a framework that describes and verifies compositionality of DSLs arose organically from the DARPA V-SPELLS program's need to correctly compose fragments of several different programming languages. While having a formalization of operads will be useful in many future contexts, we describe here our application in depth.

In Coq, we define the meta-DSL as an object using records. The underpinning idea for this choice is as follows: we want to view each DSL as an operad, so that the meta-DSL will be a subcategory of the category of operads. A further goal for this choice is that within this subcategory, we will have a natural notion of composition, and this is what we discuss in our description of the mathematical motivation for DSL composition in this context in Subsection 2.2.

Given Coq's popular usage for software verification, it was natural for the LUMOS team to choose to implement the meta-DSL in Coq; for example, Coq can be extracted into OCaml, and this can be integratd into a CI/CD pipeline. Moreover, we can identify types in a DSSM with types in Coq. These types serve as the **colors** (see Definition 3.1) for the operads that model the DSSM in Coq, and thus we choose Coq's **Type**, or a subset thereof, as the collection from which we choose our colors (see Section 4). The operad structure allows us to define functions in the DSSM with several input types and a single output type (these correspond, respectively, to $\underline{c}$ and d in Definition 3.1), and natural choices within our subcategory of operads that represent the DSSMs guarantee that functions arising from different DSLs compose correctly.

2.2 A Mathematical Model for the Syntax of DSL Composition

Our goal here is to introduce the mathematics behind the syntax of DSL composition, as well as starting to motivate our claim in Subsection 2.1 that we can view a high-level abstraction of a DSL as an operad. Moreover, we also discuss the beginnings of a constructive algebraic framework for DSL composition.

To start, we provide our informal mathematical definition of a DSL and provide an example.

Definition 2.1 *A **syntactical model for a domain-specific language** $\mathscr{D}$ is a collection of types, $\mathscr{D}_T$, and a collection of finite-arity functions, $\mathscr{D}_F$, on those types that can be composed to form new functions in $\mathscr{D}$.*

*We will often just refer to this model as a **domain-specific language**, or **DSL** for short.*

Example 2.2 *Let $\mathscr{D}_T := \{nat, str\}$, and $\mathscr{D}_F := \{$**print**, **hash**$\}$. The syntax **print** $n\,m$ takes in n,m in nat and returns the first n digits of m; **hash** str computes a hash of a given string, so that something of type nat is returned.*

*In $\mathscr{D}$, we can create the function **firstn** that prints the first n digits of a hash of a string with the syntax **print** n (**hash** str). Then **firstn** is also in $\mathscr{D}_F$.*

For clarity, in Definition 2.1, we are not only viewing a DSL $\mathscr{D}$ as a base collection of types and finite-arity functions on those types, but also as a system which allows for the creation of new functions from these base collections. Our aim with Definition 2.1 is also to describe the *syntax* of a DSL, and we will later discuss the possibility of how we can describe the semantics of a DSL using similar notions.

The main goal of the meta-DSL is to produce a a DSL $\mathscr{D}$ from a finite collection of DSLs. To provide motivation for how we are going to accomplish this in general, we provide an outline on how to accomplish this given two DSLs, $\mathscr{D}'$ and $\mathscr{D}''$.

Suppose we can regard any DSL $\mathscr{D}$ as an object in a category $\mathbb{D}$. Now given $\mathscr{D}'$ and $\mathscr{D}''$, there is a third DSL $\mathscr{Z}$ in $\mathbb{D}$, for which there are morphisms in $\mathbb{D}$:

$$\mathscr{D}' \xleftarrow{f'} \mathscr{Z} \xrightarrow{f''} \mathscr{D}''. \tag{1}$$

Then we let the **composition of $\mathscr{D}'$ and $\mathscr{D}''$ along $\mathscr{Z}$**, denoted by $\mathscr{D}$, be the object in $\mathbb{D}$ that is the **pushout** of $\mathscr{D}'$ and $\mathscr{D}''$ along $\mathscr{Z}$. This means that $\mathscr{D}$ is a DSL in $\mathbb{D}$, and there are morphisms in $\mathbb{D}$, $i' : \mathscr{D}' \to \mathscr{D}$, $i'' : \mathscr{D}'' \to \mathscr{D}$ such that the diagram in Figure 1 of the appendix in Section 7 commutes. Moreover, the triple $(\mathscr{D}, i', i'')$ is **universal** with respect to the diagram in Figure 1. This means if $(\mathscr{E}, j', j'')$ is another triple in the category $\mathbb{D}$ making the diagram in Figure 1 commute, there exists a unique $u : \mathscr{D} \to \mathscr{E}$ such that the diagram in Figure 2 of the appendix in Section 7 commutes.

A pushout is a construction that can exist in any category, and a relevant question to ask is if morphisms in 1 exist, does a pushout for $\mathscr{D}'$ and $\mathscr{D}''$ along $\mathscr{Z}$ exist in $\mathbb{D}$? In the context of our meta-DSL, it is necessary for a pushout to always exist. However, there is some generalization required since a pushout is only valid for the diagram in Figure 1.

In order to fully describe how a pushout along $\mathscr{Z}$ provides the desired definition for composition of $\mathscr{D}'$ and $\mathscr{D}''$, we first describe the construction of a pushout in the category of sets, which we denote by **Set**.

Example 2.3 *Given sets X, Y, and Z, we want to form the pushout X and Y along Z given the functions $f : Z \to X, g : Z \to Y$. The disjoint union of X and Y is given by:*

$$X \sqcup Y := \{(x,0) : x \in X\} \cup \{(y,1) : y \in Y\}. \tag{2}$$

*Here, $0,1 \in \{0,1\}$ are just regarded as distinct symbols. Then the pushout of X and Y along Z is given by the quotient of $X \sqcup Y$ by the **finest equivalence relation**, denoted by $\sim$, on 2 that identifies $(f(z),0)$ with $(g(z),1)$ for $z \in Z$. Hence the quotient, $X \sqcup Y / \sim$, is a set, and it is a straightforward exercise to show that it satisfies the properties required in Figure 2 in **Set**.*

Remark 2.4 *In Example 2.3, the **finest equivalence relation** on a set A can be defined to be an equivalence relation $\sim_R$ on A such that for any other equivalence relation $\sim_S$ on A, if $a, a' \in A$, and $a \sim_S a'$, then $a \sim_R a'$. Our aim is to make this notion constructive in the context of the category $\mathbb{D}$ by using Example 2.3 as a template. We discuss a path for this in the next example.*

Example 2.5 *Let $\mathscr{D}'$ be the DSL from Example 2.2, but with some renaming: $\mathscr{D}'_T = \{nat_0, str_0\}$ and $\mathscr{D}'_F = \{\textbf{\textit{print}}_0, \textbf{\textit{hash}}\}$.*

We let $\mathscr{D}''$ be the DSL with $\mathscr{D}''_T = \{nat_1, int, str_1\}$, and $\mathscr{D}''_F = \{\textbf{\textit{print}}_1, \textbf{\textit{add}}.nat_1, \textbf{\textit{add}}.int\}$. Ideally, the semantics of $\mathscr{D}''$ should have int as a type for the integers, $\textbf{\textit{print}}_1$ functions as $\textbf{\textit{print}}_0$ in $\mathscr{D}'$, and $\textbf{\textit{add}}.nat_1$ and $\textbf{\textit{add}}.int$ perform addition on nat_1 and int, respectively.

*The composition of $\mathscr{D}'$ and $\mathscr{D}''$ we would like would be $\mathscr{D}$, with $\mathscr{D}_T = \{nat, int, str\}$ and $\mathscr{D}_F = \{\textbf{\textit{print}}, \textbf{\textit{hash}}, \textbf{\textit{add}}.nat, \textbf{\textit{add}}.int\}$. Semantically, we want nat and str to retain all the same properties that nat_i and str_i have in $\mathscr{D}'$ and $\mathscr{D}''$, but also allow them to fit into the larger context that is their composition. Likewise, we want **print** to perform as it does in both $\mathscr{D}'$ and $\mathscr{D}''$.*

*What is a path to arrive mathematically at the syntax of the composition $\mathscr{D}$ of $\mathscr{D}'$ and $\mathscr{D}''$ in the vein of Example 2.3? A rough sketch is the following: first form the DSL $\mathscr{Z}$ for which $\mathscr{Z}_T = \{nat, str\}$, and $\mathscr{Z}_F = \{\textbf{\textit{print}}\}$. Next, construct maps between $\mathscr{Z}_T$ and $\mathscr{D}'_T$ and $\mathscr{D}''_T$ that identify nat with nat_i, and maps between $\mathscr{Z}_F$ and $\mathscr{D}'_F$ and $\mathscr{D}''_F$ that identify $\textbf{\textit{print}}_i$ with **print**. This gives us the diagram in 1, and we apply a formal construction similar to the one from Example 2.3 to the pairs $\mathscr{D}'_T, \mathscr{D}''_T$ and $\mathscr{D}'_F, \mathscr{D}''_F$ to form $\mathscr{D}_T$ and $\mathscr{D}_F$ for the DSL $\mathscr{D}$.*

Our goal is to refine the sketch in Example 2.5 into precise mathematical ideas that are constructive to dispel any ambiguity so as to allow for the formalization of a syntactical model for the composition of high-level abstractions within our meta-DSL. For example, we need to make clear what operation we are performing in lieu of a pushout in the category of sets from Example 2.3. This refinement leads to several questions:

1. How do we mathematically identify a DSL as an operad?
2. If we identify DSLs with a subcategory of operads, how do we define composition of finitely many DSLs, and does the composition always exist?
3. If composition exists, can it be made constructive and formalized?

These questions are at the intersection of our needs in LUMOS and constructive mathematics, and their answers will provide solid footing for our formalization effort. However, our focus in this paper is the formalization of operads in Coq, and we answer these questions in forthcoming work.

3 Informally Defining Operads

While there does not seem to be an agreed-upon precise definition for a **symmetric colored operad**, or **symmetric multicaegory**, we follow the definition of a symmetric colored operad in [8]. However, we note the definition in [8] does not include what is called the **equivariance axiom** in [9]; we too omit this axiom, since it is not relevant to what we want to accomplish in our work on V-SPELLS. Regardless of these distinctions, we use **operad** to mean symmetric colored operad, colored operad, or symmetric multicategory throughout this paper.

As our aim is to fully formalize the definition of an operad within Coq, we require precision, so we provide the full informal definition of an operad below in two parts. The first part consists of the data that comprises an operad.

Definition 3.1 (Data for an Operad) *An **operad** $\mathscr{O}$ consists of a collection of objects, which we will denote by T (usually referred to as **colors**), and for each $n \geq 1$, $d \in T$, and $\underline{c} := c_0, \ldots, c_{n-1}$ a sequence of objects in T, an object $\mathscr{O}\binom{d}{\underline{c}}$ in a category $\mathbb{T}$, such that*

1. *for each $d \in T$, there is a designated element $\mathbf{1}_d \in \mathscr{O}\binom{d}{d}$ called the d-**colored unit**;*
2. *if $\underline{c}'$ is a reordering of the sequence $\underline{c}$, then there is an isomorphism in the category $\mathbb{T}$:*

$$\mathscr{O}\binom{d}{\underline{c}} \cong \mathscr{O}\binom{d}{\underline{c}'};$$

3. *for any sequence $\underline{b}$ of objects in T, if we denote by $\underline{c} \bullet_i \underline{b}$ the sequence given by*

$$\underbrace{c_0, \ldots, c_{i-1}}_{\emptyset \text{ if } i=0}, \underline{b}, \underbrace{c_{i+1}, \ldots, c_{n-1}}_{\emptyset \text{ if } i=n-1},$$

then there is a function

$$\circ_i : \mathscr{O}\binom{d}{\underline{c}} \times \mathscr{O}\binom{c_i}{\underline{b}} \to \mathscr{O}\binom{d}{\underline{c} \bullet_i \underline{b}}.$$

*We call $\circ_i$ a **multi-composition operator**, and the act of using $\circ_i$ **multi-composition**.*

Remark 3.2 *In Definition 3.1, we define $\mathscr{O}\binom{d}{\underline{c}}$ to be an object in a category $\mathbb{T}$. This is the most general definition we will use, and in our examples and work, $\mathbb{T}$ will either be the category of sets or a subcategory of the category of types in a programming language.*

Example 3.3 *For a quick example of what the type signature of the multi-composition operator may look like, let $\underline{c} = c_0, c_1, c_2$; $\underline{b} = b_0, b_1$; and $i = 1$; then $\circ_1$ has type signature*

$$\mathscr{O}\begin{pmatrix} d \\ c_0, c_1, c_2 \end{pmatrix} \times \mathscr{O}\begin{pmatrix} c_1 \\ b_0, b_1 \end{pmatrix} \to \mathscr{O}\begin{pmatrix} d \\ c_0, b_0, b_1, c_2 \end{pmatrix}.$$

The data for an operad $\mathscr{O}$ in Definition 3.1 is subject to certain axiomatic constraints, and this forms the second half of our definition for an operad.

Definition 3.4 (Axiomatic Constraints for an Operad) *Let $\underline{c} := c_0, \ldots, c_{n-1}$; $\underline{b} := b_0, \ldots, b_{m-1}$; and $\underline{a} = a_0, \ldots, a_{\ell-1}$ be sequences from a collection of objects T. The axioms that the data for an operad $\mathscr{O}$ from Definition 3.1 must follow are given below.*

1. *The **horizontal associativity axiom**: Suppose $n \geq 2$ and $0 \leq i < j \leq n-1$, then for $(\alpha, \beta, \gamma) \in \mathscr{O}\binom{d}{\underline{c}} \times \mathscr{O}\binom{c_i}{\underline{a}} \times \mathscr{O}\binom{c_j}{\underline{b}}$, then in infix notation:*

$$(\alpha \circ_i \beta) \circ_{\ell-1+j} \gamma = (\alpha \circ_j \gamma) \circ_i \beta. \tag{3}$$

We visually describe this equality of terms in the commutative diagram in Figure 3.

2. *The **vertical associativity axiom**: Suppose $m,n \geq 1$, $0 \leq i \leq n-1$, and $0 \leq j \leq m-1$. Then for $(\alpha,\beta,\gamma) \in \mathscr{O}\binom{d}{\underline{c}} \times \mathscr{O}\binom{c_i}{\underline{b}} \times \mathscr{O}\binom{b_j}{\underline{a}}$, in infix notation:*

$$(\alpha \circ_i \beta) \circ_{i+j} \gamma = \alpha \circ_i (\beta \circ_j \gamma). \tag{4}$$

We visually describe this equality of terms in the commutative diagram in Figure 4.

3. *The **left unity axiom** requires that for $\alpha \in \mathscr{O}\binom{d}{\underline{c}}$ with $n \geq 1$, $\mathbf{1}_d \circ_1 \alpha = \alpha$.*

4. *The **right unity axiom** requires that for $n \geq 1$, $0 \leq i \leq n-1$, and $\alpha \in \mathscr{O}\binom{d}{\underline{c}}$, $\alpha \circ_i \mathbf{1}_{c_i} = \alpha$.*

Before we give an example, some comments are in order about Definition 3.4.

Remark 3.5

We give some insight into how we would have to handle some issues in the typing of the horizontal and vertical associativity axioms of Definition 3.4 that arise in Coq. First notice the term on the left-hand side of Equation 3 is in $\mathscr{O}\binom{d}{(\underline{c}\bullet_i\underline{a})\bullet_{\ell-1+j}\underline{b}}$, while the term on the right-hand side of Equation 3 is in $\mathscr{O}\binom{d}{(\underline{c}\bullet_j\underline{b})\bullet_i\underline{a}}$. In order for this to make since, we need to demonstrate the following equality of sequences in T:

$$(\underline{c} \bullet_i \underline{a}) \bullet_{\ell-1+j} \underline{b} = (\underline{c} \bullet_j \underline{b}) \bullet_i \underline{a}. \tag{5}$$

The left-hand side of the above equation yields:

$$\begin{aligned} (\underline{c} \bullet_i \underline{a}) \bullet_{\ell-1+j} \underline{b} &= (c_0,\ldots,c_{i-1},\underline{a},c_{i+1},\ldots,c_{n-1}) \\ &= c_0,\ldots,c_{i-1},\underline{a},c_{i+1},\ldots c_{j-1},\underline{b},c_{j+1},\ldots,c_{n-1} \end{aligned}$$

and this expression, using that $i<j$, is given by $(\underline{c} \bullet_j \underline{b}) \bullet_i \underline{a}$. In particular, we have the equality in 5. This equality provides an equality of objects in the category $\mathbb{T}$:

$$\mathscr{O}\begin{pmatrix} d \\ (\underline{c} \bullet_i \underline{a}) \bullet_{\ell-1+j} \underline{b} \end{pmatrix} = \mathscr{O}\begin{pmatrix} d \\ (\underline{c} \bullet_j \underline{b}) \bullet_i \underline{a} \end{pmatrix} \tag{6}$$

Hence in order to provide a definition in Coq for operads, we need to provide a proof that equation 5 holds for sequences in T.

A similar equality of sequences is required in our definition in Coq of the vertical associativity diagram for its use as a casting function.

While our definition seems abstract, the next example helps clarify the roots of the abstraction found in Definition 3.1 and Definition 3.4. Moreover, the next example will serve as the first application of our formal definition of operads, as we will prove in Coq that our realization of an analogous example is an operad according to our Coq specification.

Example 3.6 *If we let T be a collection of sets which is closed under taking finite products, we can define an operad $\mathbf{Sets}_T$ by letting*

$$\mathbf{Sets}_T\begin{pmatrix} d \\ c_0,\ldots,c_{n-1} \end{pmatrix} := Hom(c_0 \times \cdots \times c_{n-1}, d).$$

Here, the hom-set on the right is the collection of all functions from the product of sets $c_0 \times \cdots \times c_{n-1}$ to the set d. Given $c \in T$, the identity function on c operates as the c-colored unit in $\mathbf{Sets}_T\binom{c}{c} = Hom(c,c)$.

In this setting, we can explicitly define multi-composition $\circ_i$ from Definition 3.1 as follows: given $f \in Hom(c_0 \times \cdots \times c_{n-1}, d)$ and $g \in Hom(b_0 \times \cdots \times b_{m-1}, c_i)$, the function $f \circ_i g \in \textbf{\textit{Sets}}_T\binom{d}{\underline{c} \bullet_i \underline{b}}$ sends the $(n+m-1)$-tuple $(x_0, \ldots, x_{i-1}, \underline{y}, x_{i+1}, \ldots, x_{n-1})$ to the value $f(x_0, \ldots, x_{i-1}, g(\underline{y}), x_{i+1}, \ldots, x_{n-1})$. All other pieces of Definition 3.1 and 3.4 not mentioned above can be proved for $\textbf{\textit{Sets}}_T$ using everything defined above and basic facts in set theory.

We discuss a difference between our definition of operads and the definition found in [9].

Remark 3.7 *The definition for operads in [9] specifies that if $\underline{c} = \emptyset$, the empty sequence of symbols coming from the collection T, then the symbol $\mathscr{O}\binom{d}{\emptyset}$ still has, potentially ambiguous, meaning. Notice that in Definition 3.1 we do not allow the existence of such a symbol, since we require that the list $\underline{c}$ is not empty. Our reason for doing so is that our main application relies on giving a version of Example 3.6 in Coq. Within $\textbf{\textit{Sets}}_T$, if $\underline{c} = \emptyset$, then the product of an empty list of sets is a singleton, $\{\bullet\}$, so that $\textbf{\textit{Sets}}_T\binom{d}{\emptyset} \cong \textbf{\textit{Sets}}_T\binom{d}{\{\bullet\}}$. We can model this situation in Coq by letting $\underline{c}$ be the list whose only entry is* ***unit*** *: **Type**, which is the type in Coq with a single nullary constructor.*

Next, we describe operads for the category of types in a programming language, and we note its similarity to the Example 3.6.

Example 3.8 *Let T be a collection of types in a programming language. If $\underline{c} = c_0, \ldots, c_{n-1}$: **list** T and $d : T$, then we consider the function type*

$$\textbf{\textit{Type}}_T\binom{d}{\underline{c}} := c_0 \to \cdots \to c_{n-1} \to d. \tag{7}$$

Then terms of type $\textbf{\textit{Type}}_T\binom{d}{\underline{c}}$ are n-ary functions with codomain $\underline{c}$ and return type d. The multi-composition operator $\circ_i$ takes $f : \textbf{\textit{Type}}_T\binom{d}{\underline{c}}, g : \textbf{\textit{Type}}_T\binom{c_i}{\underline{b}}$, and returns the function $f \circ_i g : \textbf{\textit{Type}}_T\binom{d}{\underline{c} \bullet_i \underline{b}}$. Then $f \circ_i g$ acts on $x_0, \ldots, x_{i-1}, \underline{y}, x_{i+1}, \ldots, x_{n-1}$ (where $x_j : c_j$ and $y_j : b_j$) by returning $f\, x_0 \cdots x_{i-1}\, (g\, \underline{y})\, x_{i+1} \cdots x_{n-1}$. Here we forego parentheses in similarity to Coq syntax. We note the action of $\circ_i$ here is essentially a curried version of the action of $\circ_i$ in Example 3.6.

If T is the collection of all types in a given programming language, we denote this operad by ***Type****.*

4 Formally Modeling Operads in Coq

In creating a definition for the object $\mathscr{O}\binom{d}{c_0, \ldots, c_{n-1}}$ in Coq, Definition 3.1 requires that d, c_i come from a collection T. Throughout our specification in this paper, we will replace the collection of objects T with **Type**, one of Coq's in-house universes. In practice, we do need a proper subset of **Type**, but for simplicity in our paper, we use **Type**. In the event we need a restriction to a subset of **Type**, we briefly discuss how to use Tarski universes to do this after the description of our formal model in Coq.

4.1 Defining an Operad in Coq

The first goal to tackle in defining an operad in Coq is giving a definition of $\mathscr{O}\binom{d}{\underline{c}}$ that can be formalized.

Note 4.1 (A Definition for $\mathscr{O}\binom{d}{\underline{c}}$ in Coq) *Informally, we can think of $\mathscr{O}$ as a collection of sets: the $\mathscr{O}\binom{d}{\underline{c}}$. Since we want the $\mathscr{O}\binom{d}{\underline{c}}$ to be distinct, this is a collection of distinct types, so it is natural to use a record in Coq to define $\mathscr{O}$.*

We create this record in Coq, which we denote as ***Operad****, whose single field is given by a function with type signature* ***Type*** $\to$ ***list Type*** $\to$ ***Type****. An instantiation of* ***Operad*** *will yield a function* $\mathcal{O} :$ ***Type*** $\to$ ***list Type*** $\to$ ***Type****, so that* $\mathcal{O}\binom{d}{\underline{c}}$ *yields our desired object in* ***Type****. Within Coq, this means that* $f : \mathcal{O}\binom{d}{\underline{c}}$ *is equivalent to regarding* f *as a term in* $\mathcal{O}\binom{d}{\underline{c}}$.

Remark 4.2 *We note that Example 3.8 with* $T =$ ***Type*** *gives an informal definition of the main operad that we want to formalize within Coq, as well as providing a concrete example of how we will use the the definition of an operad in Coq. We denote this operad by* ***Type****.*

In the rest of our definition of an operad in Coq, we also use a record to denote the data that comprises an operad as in Definition 3.1, and the constraints the data is subject to, as in Definition 3.4. In particular, each piece in Definition 3.1 and 3.4 will be either a type in Coq that must be instantiated or a proposition in Coq that must be satisfied. We first detail how the data from Definition 3.1 will be encoded as types and propositions within Coq.

Note 4.3 (Data for an Operad in Coq) *We encode the data from Definition 3.1 as follows.*

1. *The existence of a c-colored unit in* $\mathcal{O}$ *(*1 *from Definition 3.1): for all* $c :$ ***Type****, there is a term* $\mathbf{1}_c$ *of type* $\mathcal{O}\binom{c}{c}$*;*
2. *the requirement that there is an isomorphism between* $\mathcal{O}\binom{d}{\underline{c}}$ *and* $\mathcal{O}\binom{d}{\underline{c}\sigma}$ *for a permutation* σ *on n letters in* ***Type*** *(*2 *from Definition 3.1): for all* $d :$ ***Type****,* $\underline{c}, \underline{c}' :$ ***list Type*** *with the length of* $\underline{c}$ *at least* 1 *and* $\underline{c}$ *and* $\underline{c}'$ *permutations of one another, there is a bijection between* $\mathcal{O}\binom{d}{\underline{c}}$ *and* $\mathcal{O}\binom{d}{\underline{c}'}$ *in* ***Type****;*
3. *the requirement for the existence of* $\circ_i$ *(*3 *from Definition 3.1); for all* $i, n : \mathbb{N}$*,* $d, c_i :$ ***Type****,* $\underline{b}, \underline{c} :$ ***list Type****, if* $\underline{c}$ *has length n,* $1 \leq n$*,* $i < n$*, and the nth entry of* $\underline{c}$ *is* c_i*, there is a function of type* $\mathcal{O}\binom{d}{\underline{c}} \times \mathcal{O}\binom{c_i}{\underline{b}} \to \mathcal{O}\binom{d}{\underline{c} \bullet_i \underline{b}}$.

Remark 4.4 *To make our implementation in Coq in Note 4.3 clearer, some remarks are in order about how to make the above precise within Coq.*

1. *To create a proposition that two lists,* $\underline{c}$*,* $\underline{c}'$*, are permutations of one another in Coq, we use Coq's built-in type* ***Permutation****. The proposition is written as* ***Permutation*** $\underline{c}\ \underline{c}' :$ ***Prop*** *(where* ***Prop*** *is the type of all propositions in Coq).*
2. *In* 3 *of Note 4.3, we need the use of the nth function within Coq. This function requires a default element as part of its arguments, which means we would need to choose a default element from* ***Type*** *to use consistently throughout. The choice we make in Coq is* ***unit*** $:$ ***Type****, which is the type used to represent singleton sets. However, we note the care we take with indices in all relevant proofs in our constructions to prevent the nth function from returning its default.*

We encode Definition 3.4 into Coq in a similar manner, bundling it together into a record we denote by **operadLaws**. Before we discuss the encoding of Definition 3.4, we provide a thorough and concrete example of our Coq syntax that is necessary to define 2 of Note 4.3.

Example 4.5 *We first define isomorphism in* ***Type*** *in Coq in the standard way: two types X and Y in* ***Type*** *are isomorphic if and only if there exists functions* $f : X \to Y$ *and* $g : Y \to X$ *that are inverse to one another; this is not difficult in Coq. We denote this definition by iso.*

Next, we need to define in Coq what $\mathscr{O}\binom{d}{\underline{c}}$ will actually represent as discussed in Note 4.1. First, we define the general record in Coq from Note 4.1 and simplify its notation:

Record *Operad* : ***Type*** := {*interp* : *forall* (*d* : ***Type***)(*c* : *list* ***Type***), ***Type***}.

Notation *"operad (d, c)"* := (*interp operad d c*) (*at level* 10)

Using everything we have written so far, we can make a definition for 2 in Note 4.3 in Coq.

Record *operadLaws* (*operad: Operad*) : ***Type*** :=
{*perm*: *forall* (*c c*′ : *list* ***Type***)(*d* : ***Type***),
Permutation c c′ → *iso* (*operad*(*d*, *c*)) (*operad*(*d*, *c*′))}.

This Coq syntax provides the exact proposition described in Note 4.3 and Remark 4.4. We note that in our full definition in Coq, the record ***OperadLaws*** *will also have all of the data described in Definition 3.1 and 3.4, as opposed to just the syntax above which contains only a single field.*

Next we discuss the specification of the axioms of Definition 3.4 in Coq. Here, the method of definition in Coq is similar to Example 4.5 but requires more care. This is due in part to the need to define casting functions in order to provide the proper definition in Coq. While there are four axioms, and each comes with its casting function, we present an explanation of work on the definition of the horizontal associativity axiom in Coq (1 of Definition 3.4), as it best showcases how we define these axioms in Coq.

Note 4.6 (Axioms for an Operad in Coq) *The horizontal associativity axiom in an operad (1 in Definition 3.4) can be defined in Coq by first listing a collection of parameters that we refer to as P:*

- $n, m, \ell, i, j : \mathbb{N}$;
- d, c_i, c_j : ***Type***;
- $\underline{a}, \underline{b}, \underline{c}$: ***list Type***
- $\alpha : \mathscr{O}\binom{d}{\underline{c}}, \beta : \mathscr{O}\binom{c_i}{\underline{b}}, \gamma : \mathscr{O}\binom{b_j}{\underline{a}}$
- $2 \leq n$, $1 \leq m$, *and* $1 \leq \ell$;
- $i < j$ *and* $j < n$;
- $\underline{c}$ *has length n,* $\underline{b}$ *has length m, and* $\underline{a}$ *has length* ℓ;
- *the ith entry of* $\underline{c}$ *is* c_i *and the jth entry of* $\underline{c}$ *is* c_j;

Using what is now in P, we give a proof that the ith entry of $\underline{c} \bullet_j \underline{b}$ *(see 3 of Definition 3.1 for the definition of* $\bullet$*) is* c_i*, and a proof that the* $(\ell - 1 + j)$*th entry of* $\underline{c} \bullet_i \underline{a}$ *is* c_j*; we add these proofs to P. With this update to P, we can state our formalization of the horizontal associativity axiom in Coq: for all parameters that comprise P, Equation 6 in Remark 3.5 holds, and there exists a type casting function* $C_{\textbf{h-assoc}}$ *such that*

$$C_{\textbf{h-assoc}}\, P\, ((\alpha \circ_i \beta) \circ_{\ell-1+j} \gamma) = (\alpha \circ_j \gamma) \circ_i \beta \tag{8}$$

The type-casting function $C_{\textbf{h-assoc}}$ *is necessary, since we have defined in Coq for each* d : ***Type*** *and* $\underline{c}$: ***list Type*** *that* $\mathscr{O}\binom{d}{\underline{c}}$ *be a type in* ***Type****, and the casting function provides a way of moving between the*

types in the equality of Equation 6. However, the existence of $C_{\textbf{h-assoc}}$ relies entirely on the proof of the equality of lists in Equation 5. Now the equality in Equation 5 requires a significant effort, and the most difficult part of specifying this formal model of operads in Coq of this axiom is in providing the proof of the equality of lists in Equation 5.

*Providing a formal specification of all other axioms in Definition 3.4 to be inserted into the fields of of the record **operadLaws** follows the same path as above:*

1. *carefully curate the correct collection P of parameters needed for the axiom;*
2. *add in any proofs needed that can be deduced from everything currently in P;*
3. *show that any required equality of lists holds (this will be necessary for all axioms in Definition 3.4);*
4. *create the necessary casting function.*

To give an idea of what theorems on list equality we need to prove in Coq to define the necessary casting function, we give an explicit example of the theorem on list equality needed in Coq to define the horizontal associativity axiom from Note 4.6.

Example 4.7 *Since the syntax $\underline{c} \bullet_i \underline{b}$ is just the insertion of the list $\underline{b}$ into the list $\underline{c}$ in the ith position, this is easy to define in Coq. We call this operation insert, and it has parameters i (a natural number), and lists a, b, c that are of type **list** A, for A : **Type**. With this, we can write the theorem in Coq that is the equality of lists in Equation 5.*

$$\begin{aligned}
&\textbf{Theorem}\ horizInsert : forall\ (n\,m\,\ell\,i\,j : nat)(a\,b\,c : list\,A),\\
&\quad 2 \le n \to 1 \le m \to 1 \le \ell \to i < j \to j < n \to\\
&\quad length\,c = n \to length\,a = \ell \to length\,b = m \to\\
&\quad insert\,(\ell - 1 + j)\,(insert\,i\,c\,a)\,b = insert\,i\,(insert\,j\,c\,b)\,a.
\end{aligned}$$

A proof of this theorem in Coq is not trivial, and can be found in our repository [3].

4.2 Tarski Universes

A solution to using a subset T of **Type** is to define T in Coq as a **Tarski universe**. This defines T : **Type** via an interpretation function that allows the terms of T to be regarded as symbols for types within **Type**. In this way, we can regard T as a set with an injective mapping to **Type**, which is exactly the data for a subset of **Type**. Our approach to implementing this definition in Coq involves the following:

1. a type $\mathscr{B}$ in Coq with nullary constructors, which we call the **base types**, and whose terms we refer to as **type sigils**;
2. the constructors that define the type T, which include:
   - a constructor with signature **Ty** : $\mathscr{B} \to T$ which encodes the base types into T;
   - other constructors, with signatures such as **p** : $T \to T \to T$, or **fn** : $T \to T \to T$, which will be used to model products or functions, respectively ;
3. an assignment for $\mathscr{B}$ within **Type**, and a recursively-defined interpretation function **El** : $T \to$ **Type** that assigns a value within **Type** to each $t : T$.

To help better understand this formalization, we give an example of what this would look like in Coq.

Example 4.8 *In this example, the collection of base types $\mathscr{B}$ will consist of only three type sigils: $n, b,$ and u. We define $\mathscr{B}$ in Coq as an inductive type with nullary constructors.*

```
Inductive baseTypes : Type := n | b | u.
```

We also need a an interpretation function for these sigils within Coq.

```
Definition baseInterp (b : baseTypes) : Type :=
  match b with
    | n ⇒ nat
    | b ⇒ bool
    | u ⇒ unit
  end.
```

We now use an inductive type to define the subset T of **Type** *that we will use, noting that the* ***p*** *and* ***fn*** *constructors will be used to model product and function types.*

```
Inductive baseTypes : Type :=
  | Ty : baseTypes → T
  | p : T → T → T
  | fn : T → T → T.
```

Lastly, we have to define the recursive function ***El*** *in Coq.*

```
Fixpoint El (t : T) : Type :=
  match t with
    | (Ty t) ⇒ baseInterp t
    | (p A B) ⇒ prod (El A) (El B)
    | (fn A B) ⇒ El A → El B
  end.
```

*For an explicit example, the syntax $p\,(Ty\,n)(Ty\,n) : T$ is evaluated to $nat * nat$: **Type** in Coq via* ***El****.*

5 A Proof in Coq Using Our Model

In this section, we give an implementation of the operad **Type** from Example 3.8 and show it is an operad using our formal definition in Coq of operads from Section 4. Our definition of an operad in Section 4 was a series of records, so a proof that our implementation of **Type** in Coq is an operad will consist of instantiating each field of those records.

First we discuss our implementation of **Type** in Coq, and then we show this implementation has all of the data of Definition 3.1 and Definition 3.4 by instantiating each field in the record **OperadLaws** that would occur from the information in Note 4.1, Note 4.3, and Note 4.6. We give a clear path to this below.

Remark 5.1 *To demonstrate in Coq that our implementation of* ***Type*** *is an operad, we need to:*

1. *give meaning to* $\textbf{\textit{Type}}\binom{d}{\underline{c}}$ *in Coq by defining the interpretation function in the field of the record* ***Operad*** *(see Note 4.1 and Example 4.5);*
2. *define for* c : ***Type****, the c-colored units (1 of Definition 3.1);*
3. *provide a proof that* $\textbf{\textit{Type}}\binom{d}{\underline{c}} \cong \textbf{\textit{Type}}\binom{d}{\underline{c}'}$ *for* $\underline{c}'$ *a reordering of* $\underline{c}$ *(2 of Definition 3.1);*
4. *define the multi-composition operators of* ***Type*** *in Coq (3 of Definition 3.1);*
5. *show all axioms in Definition 3.4 hold according to our definitions created in Section 4.*

Then all data from Remark 5.1 will provide an instantiation of the records **Operad** and **OperadLaws** (see Example 4.5 for the definition of these records in Coq), and this will provide a proof in Coq that **Type** is an operad.

Note 5.2 (Defining Type$\binom{d}{\underline{c}}$) *In Example 3.8, we give the definition of the required interpretation function of the record* ***Operad*** *of Note 4.1 (i.e., the function interp defined in Example 4.5) for* ***Type****: given* $\underline{c} = c_0, \ldots, c_{n-1}$: ***list*** ***Type*** *and* d : ***Type****, we write*

$$\textbf{\textit{Type}}\binom{d}{\underline{c}} := c_0 \to \cdots \to c_{n-1} \to d. \tag{9}$$

In other words, $\textbf{\textit{Type}}\binom{d}{\underline{c}}$ *is the type of n-ary functions with codomain defined by* $\underline{c}$ *and return type* d.

The right-hand side of Equation 9 can be computed via a recursive function, which we denote as ***arr*** *(short for arrow). The function* ***arr*** *has type signature* ***list*** ***Type*** $\to$ ***Type*** $\to$ ***Type****, and has* $\textbf{\textit{arr}}\,[]\,d = d$, *and* $\textbf{\textit{arr}}\,(c :: cs)\,d = c \to (\textbf{\textit{arr}}\,cs\,d)$.

We note in this definition of ***arr****, we are taking our motivation from Example 3.6 that the product of an empty collection of sets is a singleton, and there is a bijection between* $Hom(\{\bullet\}, S)$ *and* S *for any set* S *(see Remark 3.7).*

This gives 1 from Remark 5.1.

5.1 Defining the Data for the Operad Type in Coq

Next we discuss, in a series of notes, the implementation of Definition 3.1 for the operad **Type** in Coq, as well as the tools that were developed for use in this implementation.

Note 5.3 (c-colored units in Type) *If* $\underline{c}$ *has single entry* c : ***Type****, then* $\textbf{\textit{Type}}\binom{c}{\underline{c}} = c \to c$, *which is the type of all functions with codomain and domain* c. *Then* $\mathbf{1}_c := id_c$, *the identity function on* c.

Note 5.4 (Type$\binom{d}{\underline{c}} \cong$ Type$\binom{d}{\underline{c}'}$) *Our motivation is to provide an analogous isomorphism that occurs in the operad of sets,* $\textbf{\textit{Sets}}_T$. *The isomorphism, from Example 3.6, is:*

$$Hom(c_0 \times \cdots \times c_{n-1}, d) \cong Hom(c'_0 \times \cdots \times c'_{n-1}, d), \tag{10}$$

where $\underline{c} = c_0, \ldots, c_{n-1}$, *and* $\underline{c}' = c'_0, \ldots, c'_{n-1}$ *a reordering of* $\underline{c}$.*The isomorphism 10 in the context of Coq would mean we need to construct a bijection between* $\textbf{\textit{Type}}\binom{d}{\underline{c}}$ *and* $\textbf{\textit{Type}}\binom{d}{\underline{c}'}$. *Following the definition in Note 4.1 and comments in Remark 4.4, we can translate this into Coq for* ***Type*** *as: for all* d : ***Type****,* $\underline{c}, \underline{c}'$: ***list*** ***Type*** *with the length* $\underline{c}$ *at least* 1, *and* ***Permutation*** $\underline{c}\,\underline{c}'$, *there is a bijection between* $\textbf{\textit{Type}}\binom{d}{\underline{c}}, \textbf{\textit{Type}}\binom{d}{\underline{c}'}$.

We can prove this in Coq using induction on the length of $\underline{c}$, *along with some preceding lemmas about the behavior of function composition and isomorphism in* ***Type****, noting we would be using the definition of function composition and isomorphism in* ***Type*** *as discussed in Example 4.5.*

Note 5.5 ($\circ_i$ **for Type)** *Lastly, we need 3 of the definition in Note 4.3 for* ***Type*** *in Coq. Choosing to write* $\textbf{\textit{Type}}\binom{d}{c}$ *in a curried form, as opposed to using a verbatim translation of* $\textbf{\textit{Sets}}_T\binom{d}{c}$ *from Example 3.6, provides the needed flexibility, via partial application, to implement the multi-composition operator* $\circ_i$ *for* ***Type*** *in Coq. The most important piece of our definition in Coq for* $\circ_i$ *is that we define a recursive function* ***compose*** *with type signature*

$$\textbf{\textit{arr}}\,\underline{c}'\,(t \to t') \to \textbf{\textit{arr}}\,\underline{b}\,t \to \textbf{\textit{arr}}\,\underline{c}'\,(\textbf{\textit{arr}}\,\underline{b}\,t'),$$

where $t, t' :$ ***Type****, and* $\underline{c}', \underline{b} :$ ***list Type****. We defer to our repository [3] for the details of how* ***compose*** *is defined, but we notice it provides the correct type signature for* $\circ_i$*. For, if* $f :$ $\textbf{\textit{Type}}\binom{d}{\underline{c}}, g :$ $\textbf{\textit{Type}}\binom{c_i}{\underline{b}}$*, then, provided* $\underline{c}$ *is given by* $c_0, \ldots, c_{i-1}, c_i, c_{i+1}, \ldots, c_{n-1}$*, we let* $\underline{c}' = c_0, \ldots, c_{i-1}$*,* $t = c_i$*, and* $t' = c_{i+1} \to \cdots \to c_{n-1} \to d$*, then* $\textbf{\textit{compose}}\, f\, g :$ $\textbf{\textit{Type}}\binom{d}{\underline{c} \bullet_i \underline{b}}$*.*

5.2 Defining the Axioms for the Operad Type in Coq

Definition 3.1 provides the data that comprises an operad, while Definition 3.4 provides the constraints this data is subject to. We now discuss putting these constraints into Coq using the implementation for the data in Subsection 5.1.

As in the specification of the horizontal associativity axiom (1 of Definition 3.4) in Note 4.6, we keep our discussion focused on the horizontal associativity axiom, as the proof that our implementation of **Type** satisfies all other axioms in Coq follows from similar arguments.

We define the horizontal associativity axiom for **Type** as a proposition using the definition of the horizontal associativity axiom in Coq from Note 4.6. In this situation, we are rephrasing Equation 3 using the **compose** function from Note 5.5.

Our first hurdle in this direction comes from noticing that Coq does not automatically recognize the equality of types,

$$\mathbf{arr}\,\underline{c}'\,(\mathbf{arr}\,\underline{b}\,t) = \mathbf{arr}\,(\underline{c} \bullet_i \underline{b})\,d. \tag{11}$$

Where, $\underline{c} = c_0, \ldots, c_{i-1}, c_i, c_{i+1}, \ldots, c_{n-1}$, $\underline{c}' = c_0, \ldots, c_{i-1}$, and $t = c_{i+1} \to \cdots \to c_{n-1} \to d$, so that, as in 3 of Definition 3.1, $\underline{c} \bullet_i b = c_0, \ldots, c_{i-1}, \underline{b}, c_{i+1}, \ldots, c_{n-1}$.

In Coq, we are taking $f :$ $\mathbf{arr}\,\underline{c}\,d$ and $g :$ $\mathbf{arr}\,\underline{b}\,c_i$, so that **compose** $f\,g$ has type given by the left-hand side of Equation 11. However, in our definition of operads in Coq, we define $f \circ_i g$ (which is represented by **compose** $f\,g$) to have type $\mathbf{Type}\binom{d}{\underline{c} \bullet_i \underline{b}}$, which is given by the right-hand side of Equation 11, hence this presents an immediate problem. As in Note 4.6, our solution here is through type casting.

Much of the proof that our implementation of **Type** in Coq is an operad according to our specification is accomplished through the use of type casting functions, and we now discuss the tools we developed in this direction. First, we give the definition we use for a type cast within Coq, and then supply the Coq syntax for it.

Definition 5.6 *Given* $A, B :$ ***Type****, and an equation,* $A = B$*, a* ***type cast****,* $\mathscr{C}_{A=B}$*, is a function such that for* $a : A$*,* $\mathscr{C}_{A=B}\, a : B$*.*

We easily define a type cast in Coq as function that takes as parameters the type equation $A = B$, for $A, B :$ **Type**, and a term $e : A$. In order to manipulate the type casts that occur throughout our proof that **Type** satisfies the formalization of Definition 3.4, we prove a handful of general facts about type casts in Coq, which we discuss below.

Note 5.7

1. *The composition two type casts is a type cast: given equations of types $A = B$ and $B = C$, then $\mathscr{C}_{B=C} \circ \mathscr{C}_{A=B} = \mathscr{C}_{A=C}$.*
2. *A type cast using an equation of types $A = A$ (i.e., a type cast between two types Coq recognizes as identical) is equal to the identity: $\mathscr{C}_{A=A}\, a = a$ for $a : A$.*
3. *Two type casts between the same two types (i.e., both using equations of type $A = B$) are equal: for all $a : A$, $\mathscr{C}_{A=B}\, a = \mathscr{C}'_{A=B}\, a$.*
4. *Given a type equality $B = C$, we also have another type equality, $A \to B = A \to C$. In particular, we have two associated type casts: $\mathscr{C}_{A\to B=A\to C}$ and $\mathscr{C}_{B=C}$. If $f : A \to B$ and $a : A$, then $(\mathscr{C}_{A\to B=A\to C}\, f)\, a = \mathscr{C}_{B=C}\, (f\, a)$.*

The statements of Note 5.7 smooth the process of showing **Type** satisfies the formalization of Definition 3.4, as this involves manipulations of several type casts. For example, 2 and 3 of Note 5.7 ensure that it is not necessary to keep track of how these manipulations impact the equations on which the type casts rely, since we need only that the types involved match in order to show equality.

Proving the statements in Note 5.7 are not difficult, and we provide the syntax of our proof for 2 of Note 5.7 as an example.

Example 5.8

Lemma *typecastSelf* $\{A : \textit{Type}\}\, (eq : A = A)\, (x : A) :$

typecast $eq\, x = x$.

Proof.

unfold typecast.

rewrite $\leftarrow$ *Eqdep.Eq_rect_eq.eq_rect_eq.*

reflexivity.

Qed.

The proof is relatively straightforward, and relies on Coq's `eq_rect_eq` *axiom, which is true of the proofs of all statements in Note 5.7 except 1. The proofs of the other statements in Note 5.7 are proved similarly.*

Now we discuss how to utilize the statements of Note 5.7 in the proof that the horizontal associativity axiom (1 of Definition 3.4) is satisfied by our implementation of **Type** in Coq. Our first step is to prove the following key equality involving the **compose** function. We want to show that

$$\textbf{compose}\, (\mathscr{C}\, (\textbf{compose}\, (\mathscr{C}\, \alpha)\, \beta))\, \gamma \tag{12}$$

is equal to

$$\mathscr{C}\, (\textbf{compose}\, (\mathscr{C}\, (\textbf{compose}\, (\mathscr{C}\, \alpha)\, \gamma))\, \beta). \tag{13}$$

Here, α, β, γ are terms of the appropriate types in **Type**, and the equations that decorate type casts have been suppressed from the notation. If we step back for a moment and compare the terms in 12 and 13 to the terms in Equation 5 for the horizontal associativity axiom, we notice that 12 is representing the

right-hand side of Equation 5, and 13 the left-hand side of Equation 5. To make clear why, recall that **compose** formalizes the multi-composition operator $\circ_i$, and that if all type casts are ignored in 12 and 13, then we arrive at the terms on both sides of Equation 5.

We can show equality between the terms in 12 and 13 by induction on the appropriate lists and several uses of 4 of Note 5.7. The remainder of the proof that the formalization of the horizontal associativity axiom holds in our implementation of **Type** requires more manipulations of type casts using the statements of Note 5.7.

Demonstrating that the remainder of the formalized axioms of Definition 3.4 holds for our implementation of **Type** in Coq follows a similar pattern: translate the formalization of the axiom in our specification to an appropriate equality involving the use of the **compose** function and suitable type casts, and show this equality holds by using the statements on type casts in Note 5.7.

Remark 5.9 *Our initial aim was to use operads to mathematically specify syntactical models of domain-specific languages, so that we can also mathematically define composition of these syntactical models. In this vein, the denotational semantics of the meta-DSL had been cast aside in this effort. However, we note that the proof built in this section constructed an operad that has denotational semantics by viture of being defined in Coq. In particular, this operad lives in the Calculus of Inductive Constructions and a more general DSL can be built from the terms from this operad.*

6 Related Work

In [6], the authors present a formalization of a simpler type of an operad using Cubical Agda, which is an extension of Agda with Cubical Type Theory. Cubical Type Theory is an alternative to Homotopy Type Theory that is more directly amenable to constructive interpretations, so fully understanding their specification of operads in [6] requires a working knowledge of a variant of Homotopy Type Theory, as well as knowledge of Agda.

Moreover, Agda does not have significant automation, so showing, for example, our proof in Section 5 would require significantly more work. Nevertheless, we believe it would be possible to translate our work into Agda, albeit with much more boilerplate code (e.g., handwritten structural induction tactics).

We also compare what was formalized in our work to that of [6]. The work in [6] uses an **operad** in the same sense we do, but its set of colors T always has $|T| = 1$. In particular, $\mathscr{O}\binom{d}{\underline{c}}$ can be parametrized by the natural numbers, so we can write $\mathscr{P}(n) := \mathscr{O}\binom{d}{\underline{c}}$, if $|\underline{c}| = n$, where $\underline{c}$ has $c_i = d$ for all i. There is also a unique identity, $\mathbf{1} \in \mathscr{P}(1)$); the functions $\circ_i$ have type signature $\mathscr{P}(n) \times \mathscr{P}(m) \to \mathscr{P}(n+m-1)$; and there is a significant simplification of the associativity axioms (1 and 2 of Definition 3.4). We also note [6] defines its operads as having the **equivariance axiom** given in [9], which we did not include since it was unncessary for our applications in V-SPELLS.

References

[1] John C. Baez & John Foley (2020): *Operads for Designing Systems of Systems*. *CoRR* abs/2009.12647. arXiv:2009.12647.

[2] John C. Baez & Nina Otter (2017): *Operads and phylogenetic trees*. *Theory Appl. Categ.* 32, pp. Paper No. 40, 1397–1453.

[3] Zachary Flores, Angelo Taranto, Eric Bond & Yakir Forman (2023): *coq-operads package*. Available at https://github.com/twosixlabs/coq-operads/.

[4] John D. Foley, Spencer Breiner, Eswaran Subrahmanian & John M. Dusel (2021): *Operads for complex system design specification, analysis and synthesis*. *CoRR* abs/2101.11115. arXiv:2101.11115.

[5] Samuele Giraudo, Jean-Gabriel Luque, Ludovic Mignot & Florent Nicart (2016): *Operads, quasiorders, and regular languages*. *Adv. Appl. Math.* 75, pp. 56–93, doi:10.1016/j.aam.2016.01.002. Available at https://doi.org/10.1016/j.aam.2016.01.002.

[6] Brandon Hewer & Graham Hutton (2023): *HoTT Operads*. *Symposium on Principles of Programming Languages*. Available at http://www.cs.nott.ac.uk/~pszgmh/operads.pdf.

[7] Sophie Libkind, Andrew Baas, Evan Patterson & James Fairbanks (2022): *Operadic Modeling of Dynamical Systems: Mathematics and Computation*. *Electronic Proceedings in Theoretical Computer Science* 372, pp. 192–206, doi:10.4204/eptcs.372.14.

[8] David Spivak (2013): *The Operad of Wiring Diagrams: Formalizing a Graphical Language For Databases, Recursion, and Plug-and-Play Circuits*. arXiv:1305.0297v1.

[9] Donald Yau (2018): *Operads of wiring diagrams*. *Lecture Notes in Mathematics* 2192, Springer, Cham, doi:10.1007/978-3-319-95001-3. Available at https://doi.org/10.1007/978-3-319-95001-3.

7 Appendix

In this appendix, we put figures that are helpful for understanding the main body of the paper and easily referenced.

7.1 Diagrams for Pushouts

The commutative diagram for a pushout $\mathscr{D}$ of a diagram 1 in the category $\mathbb{D}$.

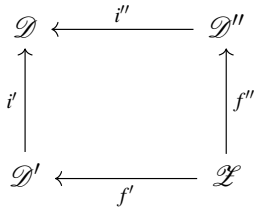

Figure 1: A pushout in $\mathbb{D}$

The commutative diagram for the universality property that a pushout $\mathscr{D}$ of a diagram 1 must satisfy in the caegory $\mathbb{D}$.

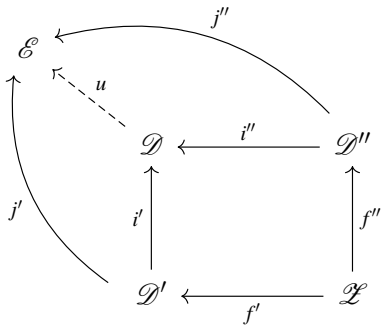

Figure 2: Universality of a pushout in $\mathbb{D}$

7.2 Diagrams for Associativity Axioms in Operads

Here, we present commutative diagrams in Definition 3.4 that are not necessary to the immediate exposition in that section, but help us understand what Equation 3 and Equation 4 are communicating.

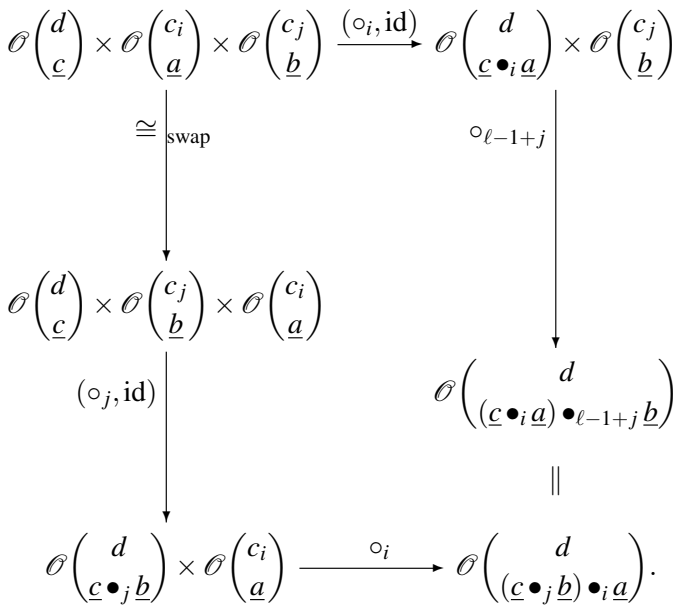

Figure 3: The horizontal associativity axiom

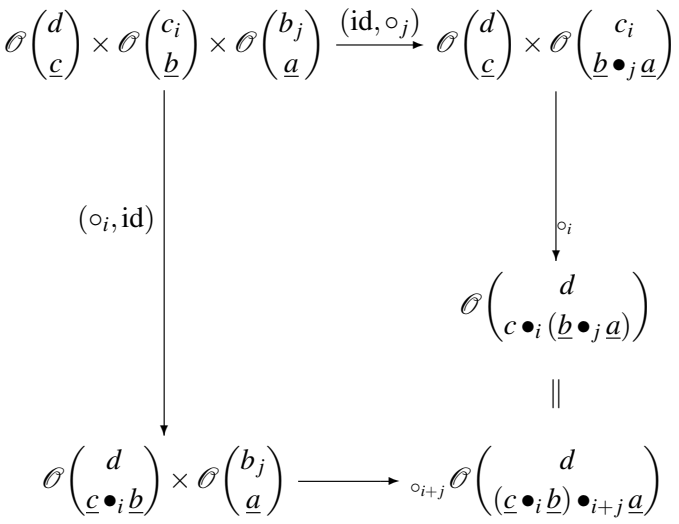

Figure 4: The vertical associativity axiom