HoTT Operads

Brandon Hewer
School of Computer Science, University of Nottingham, UK

Graham Hutton
School of Computer Science, University of Nottingham, UK

Abstract

Internalising classes of datatypes has been a longstanding pursuit in type theory. For example, containers capture strictly positive types, while combinatorial species capture finitely labelled structures. To date, however, there has been no similar attempt to internalise classes of *operations* on datatypes. In this paper we show how the theory of operads, which extend species with a well-behaved notion of composition, provides a natural approach to internalising finitary operations. We present an internalisation of a generalised notion of operad in homotopy type theory, which provides a generic framework for capturing and reasoning about operations with particular algebraic properties. All our results are formalised in Cubical Agda.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Constructive mathematics

Keywords and phrases operads, homotopy type theory, proof assistants

Digital Object Identifier 10.4230/LIPIcs.TYPES.2025.4

Supplementary Material *Software (Formalisation)*: `https://github.com/brandonhewer/Operads-HoTT`

Funding This work was partially funded by EPSRC grant EP/Y010744/1, *Semantics-Directed Compiler Construction: From Formal Semantics to Certified Compilers*.

Acknowledgements The authors thank the reviewers for their constructive feedback and suggestions that have helped to improve this article.

1 Introduction

Type theorists have a longstanding interest in studying classes of datatypes. For example, containers [1] provide an internal theory of datatypes that captures strictly positive types [2], while combinatorial species capture finitely labelled structures [26]. Whereas containers represent datatypes by their "shapes" and "positions", species describe the construction of a structure from a finite set of labels. One of the key applications of both containers and species is generic programming in a dependent type theory. In this way, both theories give rise to an internalised calculus of datatypes.

It is also natural to consider a calculus of *operations over datatypes*. Surprisingly, this idea has been much less explored in type theory. However, category theory provides a well-known tool for describing such algebraic structures. In particular, the notion of an *operad* generalises the concept of a category to a "multicategory" (with one object), where the source of a morphism is a finite sequence of objects. The theory of operads can be viewed as an extension of the theory of combinatorial species, in which an operad is simply a species together with a well-behaved notion of composition.

In this paper we present an internalised calculus of composable operations by realising the categorical notion of an operad in a suitable intensional type theory. The key contribution of the paper is the presentation of a generalised notion of operad in homotopy type theory (HoTT) that allows for different equational theories to be captured by varying the groupoid on which operations are indexed. More concretely, we make the following contributions:

- Internalising the notion of planar (section 4) and symmetric (section 5) operads in HoTT, and providing practical examples of each form;
- Introducing a generalised notion of operad whose operations are indexed by an arbitrary groupoid (section 6), and proving that such operads form a category (section 7);
- Proving that every such operad induces a canonical monad (section 8), which provides an *operadic* style of constructing programs from a small collection of composable terms;
- Showing how the free operad can be constructed as a higher inductive family (section 9).

All our results are formalised in Cubical Agda [4] version 0.9. The paper is aimed at a general type theory audience who are interested in investigating operadic structures, and we provide an introduction to the required additional concepts from cubical type theory in Section 2.

2 Background and Metatheory

In this paper, ideas and formalisations are presented in the metatheory of Cubical Agda [23]. We begin by reviewing *path types*, a key concept in HoTT. For any type A, we can construct the type $x \equiv y$ of *paths* between terms $x, y : A$. The underlying definition of a path type varies between different models of HoTT. For example, in the CCHM model of cubical type theory [12], on which Cubical Agda is based, a path type on elements of A corresponds to a continuous function from the real interval $[0, 1]$ to A.

Readers unfamiliar with HoTT can think of the path type $x \equiv y$ as behaving identically to the inductively defined identity type. In particular, the eliminator for path types is given by the J-rule, which states that for any term $x : A$ and family of types $M : (y : A) \to x \equiv y \to \mathrm{Type}$, if we have a proof $t : M\ x\ \mathbf{refl}$ then for all $y : A$ and $p : x \equiv y$ we can construct a term $J_{M,t}\ x\ y\ p : M\ y\ p$. Intuitively, the J-rule states that if the end-point y of the path p can vary, then we can substitute p for reflexivity at x.

Cubical Agda also provides a primitive construction for heterogeneous path types which are presented in the form $\mathsf{PathP}\ (\lambda\ i \to T)\ a\ b$ for terms $a : A$, $b : B$ and $T : \mathsf{Type}$. The first argument to PathP is a continuous function out of the built-in interval type I for which the endpoints $\mathsf{i0}\ \mathsf{i1} : \mathsf{I}$ are mapped to a and b respectively.

3 Basic Idea

In this section we introduce the notion of an operation and a collection of operators using a simple example, and describe what it means for a collection of operations to be operadic.

Informally, we can think of an *operation* as a map from elements of a structured collection to an element of the same collection. For example, addition of natural numbers and disjoint union of finite sets are examples of *binary* operations, whose domain is given by the binary product of the underlying collection. More generally, we will consider any domain that can be expressed as a strictly positive endofunctor on the underlying collection, e.g. finite and countable products. That is, we can understand an operation on a type A to simply be given by a strictly positive endofunctor $F : \mathsf{Type} \to \mathsf{Type}$, together with an F-algebra $f : F\ A \to A$. For example, consider the following simple expression language in Agda:

```
data Expr : Type where
  val : ℕ → Expr
  add : Expr → Expr → Expr
```

The constructors val and add can be seen as operations on Expr: val is an algebra for the constant functor choosing $\mathbb{N}$, while add is a (curried) algebra for the binary diagonal functor.

While the above definition of an operation is sufficient to discuss properties of operations in isolation, it is not sufficient to describe how operations interact, and in particular compose. For example, consider the collection of operations constructed as finite compositions of val and add, together with the identity function id : Expr → Expr. Importantly, the inclusion of id allows us to introduce variables, e.g. in the term $\lambda\ x\ y\ z \to$ add (add x (val 1)) (add y z). Operations constructed in this way are *finite* and *linear*, where *linear* means that each variable bound by the surrounding λ appears exactly once in the body of the operation.

To represent the desired collection of operations on Expr, we first observe that the type $\sum[\ n \in \mathbb{N}\ ]\ ((\mathsf{Fin}\ n \to \mathsf{Expr}) \to \mathsf{Expr})$, where Fin : $\mathbb{N}$ → Type is the family of totally-ordered finite sets, corresponds to all (non-symmetric) finitary operations on Expr. However, our goal is to capture only those operations constructible as compositions of val, add and id. In order to do this, we might search for a subtype of all finitary operations which precisely corresponds to such a collection. Alternatively, it is perhaps easier and more natural to give an abstract representation of this collection, together with an interpretation as concrete operations. Indeed, we can do this by first defining the following inductive family:

```
data IExpr : ℕ → Type where
  id↑ : IExpr 1
  val↑ : ℕ → IExpr 0
  add↑ : IExpr m → IExpr n → IExpr (m + n)
```

For every n : $\mathbb{N}$, we can think of IExpr n as representing the n-ary operations that are constructible as finite compositions of id, val and add. In particular, id↑ and val↑ correspond directly to id and val, while add↑ describes how the outputs of an m-ary and n-ary operation can be plugged into the inputs of add to construct an $(m + n)$-ary operation. To define a function ⟦_⟧ that interprets abstract terms of IExpr n as concrete n-ary functions, we use projections π^1 : (Fin $(m + n)$ → Expr) → Fin m → Expr and π^2 : (Fin $(m + n)$ → Expr) → Fin n → Expr, which project the first m and last n elements from an $(m + n)$-fold product:

```
⟦_⟧ : IExpr n → (Fin n → Expr) → Expr
⟦ id↑ ⟧ es = es zero
⟦ val↑ n ⟧ es = val n
⟦ add↑ e₁ e₂ ⟧ es = add (⟦ e₁ ⟧ (π¹ es)) (⟦ e₂ ⟧ (π² es))
```

While we do not give a proof here, it can be shown that ⟦_⟧ is injective, and as the codomain (Fin n → Expr) → Expr is an h-set, it is also an embedding. Consequently, we can understand IExpr n to be a subtype of (Fin n → Expr) → Expr, and indeed up to equivalence it is precisely the type of operations that are constructible as finite compositions of add, val and id.

It is natural to identify a reasonable condition for when a countable family such as IExpr : $\mathbb{N}$ → Type can be considered an abstract collection of finitary operations without appealing to a specific interpretation function such as ⟦_⟧. Importantly, this condition should capture the compositional structure of the operations. The minimal such condition we will require is for the family to be equipped with an identity operation and closed under an n-ary composition map that respects identity and associativity laws. To give a definition of an n-ary composition map, we first require a family of functions sum n : (Fin n → $\mathbb{N}$) → $\mathbb{N}$ for calculating the sum of n natural numbers, which can be defined by induction on n in the obvious way. For every n : $\mathbb{N}$ and n-ary product of natural numbers ns : Fin n → $\mathbb{N}$, we can now construct a family of functions comp n ns : IExpr n → ((i : Fin n) → IExpr $(ns\ i)$) → IExpr (sum n ns):

```
comp 1 ns id↑ es = es zero
comp 0 ns (val↑ k) es = val↑ k
comp .(m + n) ns (add↑ e₁ e₂) es =
  let es₁ = comp m (π¹ ns) e₁ (Π¹ es)
      es₂ = comp m (π² ns) e₂ (Π² es)
   in subst IExpr (π¹+π² ns) (add↑ es₁ es₂)
```

In the above definition, we use the n-ary dependent projection functions Π^1 and Π^2, together with the path $\pi^1{+}\pi^2\ ns : \mathsf{sum}\ m\ (\pi^1\ ns) + \mathsf{sum}\ n\ (\pi^2\ ns) \equiv \mathsf{sum}\ (m+n)\ ns$, whose constructions can be found in our Cubical Agda formalisation.

Intuitively, our composition map plugs the outputs of n operations into the inputs of an n-ary operation. Importantly, we can show that comp preserves associativity and identity laws, where the identity operation is given by id↑. The left and right identity laws assert that for any n-ary operation, plugging its output into the input of the identity operation, or plugging the identity's output into each of its n inputs, leaves the operation unchanged. Associativity states that composing a finitely branching tree of operations "bottom-up" is the same as composing "top-down". We formally characterise these laws in Section 4.

We can also observe that a similar compositional structure to that described by comp exists for the family of all finitary operations $(\mathsf{Fin}\ n \to \mathsf{Expr}) \to \mathsf{Expr}$, where the composition map is simply given by n-ary function composition, and the unit operation is given by the term $\lambda\ es \to es\ \mathsf{zero} : (\mathsf{Fin}\ 1 \to \mathsf{Expr}) \to \mathsf{Expr}$. The interpretation function $[\![\_]\!] : \mathsf{IExpr}\ n \to (\mathsf{Fin}\ n \to \mathsf{Expr}) \to \mathsf{Expr}$ can then be understood as a homomorphism between such structures, respecting both the composition map and the identity operation. This common compositional structure on collections of operations, is precisely the structure described by planar operads.

4 Planar Operads

A collection of finitary operations with a distinguished unit, closed under an n-ary composition map that respects unitality and associativity, is known as a *planar operad*. For example, the compositional structure for IExpr in the previous section describes a planar operad. In particular, planar here simply means that our operadic composition map has a total order on input operations. Consequently, the composition map of a planar operad does not necessarily respect reordering of the input operations. In this section we formalise planar operads in Cubical Agda, and provide another example and important properties of our definition.

We begin by recalling that a countable family of types $K : \mathbb{N} \to \mathsf{Type}$ can be understood as an *arity-indexed family of types of operations*: for each $n : \mathbb{N}$, an element of $K\ n$ represents an abstract n-ary operation, without yet committing to its action on data. The operadic structure on K then specifies how such operations compose. To avoid higher coherences, we consider only families of h-sets, i.e. $K : \mathbb{N} \to \mathsf{hSet}$ where hSet is the universe of h-sets. In Cubical Agda, hSet is not a built-in universe but is expressed as a dependent sum $\sum[\ A \in \mathsf{Type}\ ]\ \mathsf{isSet}\ A$, where $\mathsf{isSet}\ A$ is the proposition that A is an h-set. However, given a family of sets $K : \mathbb{N} \to \mathsf{hSet}$, we will write $K\ n : \mathsf{Type}$ for the first projection, and make it clear when we use the proof that this type is an h-set.

▶ **Definition 1** (Planar operad). *Given any family of h-sets $K : \mathbb{N} \to \mathsf{hSet}$, we can define the planar operads on K as a record type $\mathsf{record}\ \mathsf{NonSymmOperad}\ K : \mathsf{Type}$, i.e. a finitely iterated sigma type with named projections. In particular, a planar operad $O : \mathsf{NonSymmOperad}\ K$ has a distinguished identity operation $\mathsf{id}\ O : K\ 1$ and a family of composition maps*

$$\mathsf{comp}\ O\ n\ ns : K\ n \to ((i : \mathsf{Fin}\ n) \to K\ (ns\ i)) \to K\ (\mathsf{sum}\ n\ ns)\,,$$

for every $n : \mathbb{N}$ *and* $ns : \mathit{Fin}\ n \to \mathbb{N}$*. The composition map tells us how to plug the outputs of* n *operations, each with its own number of inputs* ns*, into the inputs of an operation with* n *inputs. Furthermore, we require paths witnessing that this composition map respects the identity and associativity laws. In particular, the left identity law is witnessed by a path*

$$\mathit{idl}\ O\ n\ k : \mathit{comp}\ O\ 1\ (\lambda\ i \to n)\ (\mathit{id}\ O)\ (\lambda\ i \to k) \equiv k,$$

for every $n : \mathbb{N}$ *and* $k : K\ n$*. The path* $\mathit{idl}\ O\ n\ k$ *witnesses that plugging the output of* k *into the identity is equal to* k*. Right identity is then witnessed by a heterogeneous path*

$$\mathit{idr}\ O\ n\ k : \mathit{PathP}\ (\lambda\ i \to K\ (\mathit{sum\text{-}idr}\ n\ i))\ (\mathit{comp}\ O\ n\ (\lambda\ i \to 1)\ k\ (\lambda\ i \to \mathit{id}\ O))\ k,$$

where $\mathit{sum\text{-}idr}\ n : \mathit{sum}\ n\ (\lambda\ i \to 1) \equiv n$ *witnesses that the sum of* n *ones is* n*. Intuitively,* $\mathit{idr}\ O\ n\ k$ *witnesses that the operation plugging the output of the identity into each of the* n *inputs of* k *is equal to* k*. Finally, in order to state the operad associativity law, we first introduce the following family of equivalences,*

$$\mathit{sum}\Sigma\ ns : \mathit{Fin}\ (\mathit{sum}\ n\ ns) \to \Sigma[\ i \in \mathit{Fin}\ n\ ]\ \mathit{Fin}\ (ns\ i)$$

for all $n : \mathbb{N}$*,* $ns : \mathit{Fin}\ n \to \mathbb{N}$*. This family witnesses that totally ordered finite sets are closed under dependent sums, and can be used to define the associativity condition for* sum*,*

$$\begin{aligned}\mathit{sum\text{-}assoc}\ n\ ns\ nss : \mathit{sum}\ n\ (\lambda\ i \to \mathit{sum}\ (ns\ i)\ (nss\ i))\\ \equiv \mathit{sum}\ (\mathit{sum}\ n\ ns)\ (\mathit{uncurry}\ nss \circ \mathit{sum}\Sigma)\end{aligned}$$

for all $nss : (i : \mathit{Fin}\ n) \to \mathit{Fin}\ (ns\ i) \to \mathbb{N}$*. The operad associativity law,* $\mathit{assoc}\ O\ n\ ns\ nss$*, can then be expressed as a heterogeneous path over* $\mathit{cong}\ K\ (\mathit{sum\text{-}assoc}\ n\ ns\ nss)$ *between top-down composition*

$$\mathit{comp}\ O\ n\ (\lambda\ i \to \mathit{sum}\ (ns\ i)\ (nss\ i))\ (\lambda\ i \to \mathit{comp}\ O\ (ns\ i)\ (nss\ i)\ (ks\ i)\ (kss\ i))$$

$$\begin{aligned}&\mathit{comp}\ O\ n\ (\mathit{sum}\ n\ ns)\ (\mathit{uncurry}\ nss \circ \mathit{sum}\Sigma\ ns)\\ &\qquad(\mathit{comp}\ O\ n\ ns\ k\ ks)\ (\mathit{uncurry}\ kss \circ \mathit{sum}\Sigma\ ns)\end{aligned}$$

Beyond the IExpr operad of Section 3, a second example is given by *partial lists* with "holes".

▶ **Example 2** (Partial-list planar operad)**.** We define the family of partial lists with "holes" as the following inductive family:

```
data PartialList (A : Type) : ℕ → Type where
  []  : PartialList A 0
  _::_ : A → PartialList A n → PartialList A n
  poke : PartialList A n → PartialList A (suc n)
```

Intuitively, the family PartialList A corresponds to partial lists indexed by their number of holes. The constructors [] and _::_ are as usual for lists, while poke introduces a hole at the start of a list. Partial lists have a concatenation operation _++_ : PartialList $A\ m \to$ PartialList $A\ n \to$ PartialList $A\ (m + n)$, defined by:

```
[] ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
poke xs ++ ys = poke (xs ++ ys)
```

Given any h-set A : Type, the countable family of h-sets PartialList A comes equipped with a planar operadic structure

$$O : \mathsf{NonSymmOperad}\ (\mathsf{PartialList}\ A)$$

with n-ary composition map defined as follows:

$$\begin{aligned}
&\mathsf{comp}\ O\ 0\ ns\ xs\ xss = xs\\
&\mathsf{comp}\ O\ (\mathsf{suc}\ n)\ ns\ (x :: xs)\ xss = x :: \mathsf{comp}\ O\ (\mathsf{suc}\ n)\ ns\ xs\ xss\\
&\mathsf{comp}\ O\ 1\ ns\ (\mathsf{poke}\ xs)\ xss = \mathsf{subst}\ (\mathsf{PartialList}\ A)\ (+\text{-}\mathsf{zero}\ (ns\ \mathsf{zero}))\ (xss\ \mathsf{zero} +\!\!+ xs)\\
&\mathsf{comp}\ O\ (\mathsf{suc}\ (\mathsf{suc}\ n))\ ns\ (\mathsf{poke}\ xs)\ xss = xss\ \mathsf{zero} +\!\!+ \mathsf{comp}\ O\ (\mathsf{suc}\ n)\ (ns \circ \mathsf{suc})\ xs\ (xss \circ \mathsf{suc})
\end{aligned}$$

The identity operation for the compositional structure on partial lists is then simply $\mathsf{id}\ O = \mathsf{poke}\ []$. The identity and associativity laws are proved in our Cubical Agda formalisation.

We might ask if the operations characterised by the operadic structure on PartialList A correspond to a collection of concrete n-ary operations. Indeed, there is an interpretation $\mathsf{fill} : \mathsf{PartialList}\ A\ n \to (\mathsf{Fin}\ n \to \mathsf{List}\ A) \to \mathsf{List}\ A$ which fills n holes with n lists.

▶ **Proposition 3** (fill is a planar operad homomorphism). *Similarly to* $[\![\_]\!] : \mathit{IExpr}\ n \to (\mathit{Fin}\ n \to \mathit{Expr}) \to \mathit{Expr}$ *in Section 3,* fill *is a homomorphism between planar operads and respects the compositional structure and identity operation. The proof can be found in our Cubical Agda formalisation, and we introduce operad morphisms and algebras in Section 7.*

5 Symmetric Operads

In the previous section we introduced planar operads in HoTT. These generalise the usual notion of operad, whose operations also come equipped with actions of symmetric groups compatible with the composition map. In this section, we show how our notion of a planar operad can be specialised to the symmetric case. Naively, this means extending our definition of a planar operad O : NonSymmOperad K with a field

$$\mathsf{permute}\ O\ n : \mathsf{Fin}\ n \simeq \mathsf{Fin}\ n \to K\ n \simeq K\ n,$$

for every $n : \mathbb{N}$, where $A \simeq B$ is the type of equivalences between the h-sets A and B. An equivalence $\sigma : A \simeq B$ has projections $\mathsf{fun}\ \sigma : A \to B$, $\mathsf{inv}\ \sigma : B \to A$, $\mathsf{linv} : (a : A) \to \mathsf{inv}\ \sigma\ (\mathsf{fun}\ \sigma\ a) \equiv a$ and $\mathsf{rinv} : (b : B) \to \mathsf{fun}\ \sigma\ (\mathsf{inv}\ \sigma\ b) \equiv b$. Moreover, permute $O\ n$ must be a group homomorphism between the automorphism groups $\mathsf{Fin}\ n \simeq \mathsf{Fin}\ n$ and $K\ n \simeq K\ n$, but we omit the evident fields to witness preservation of identity, composition and symmetry. Furthermore, for all $ns : \mathsf{Fin}\ n \to \mathbb{N}$, $k : K\ n$, $ks : (i : \mathsf{Fin}\ n) \to K\ (ns\ i)$ and $\sigma : \mathsf{Fin}\ n \simeq \mathsf{Fin}\ n$, we require a field witnessing that permute $O\ n$ is compatible with the operad composition map. That is, this field asserts the operadic *equivariance* law, which states that the symmetric-group action commutes with composition. We retain the name symm to mirror our formalisation:

$$\begin{aligned}
&\mathsf{symm}\ O\ n\ ns\ k\ ks\ \sigma : \mathsf{PathP}\ (\lambda\ i \to K\ (\mathsf{sum\text{-}permute}\ n\ ns\ \sigma\ i))\\
&\qquad (\mathsf{comp}\ O\ n\ ns\ (\mathsf{fun}\ (\mathsf{permute}\ O\ n\ \sigma)\ k)\ ks)\\
&\qquad (\mathsf{comp}\ O\ n\ ns\ k\ (ks \circ \mathsf{fun}\ \sigma))
\end{aligned}$$

This naive presentation of symmetric operads can be internalised in standard Martin-Löf type theory, but requires an explicit construction of the permutation homomorphism and proof of compatibility with the operad composition map. However, this approach can be

unwieldy in practice. This problem arises because a countable family of h-sets $K : \mathbb{N} \to \mathsf{Type}$ does not always come equipped with actions of symmetric groups. Intuitively, this means that the composition map of a planar operad can freely make use of the ordering of operations. This is an expected consequence of indexing operations by the natural numbers, which are in equivalence with the type of finite sets equipped with a total order, i.e. the type $\sum[\ A \in \mathsf{Type}\ ]\ \sum[\ n \in \mathbb{N}\ ]\ A \simeq \mathsf{Fin}\ n$. This can readily be seen by recognising that for any $n : \mathbb{N}$, the type $\sum[\ A \in \mathsf{Type}\ ]\ A \simeq \mathsf{Fin}\ n$ is, by univalence, equivalent to the singleton type $\sum[\ A \in \mathsf{Type}\ ]\ A \equiv \mathsf{Fin}\ n$ centred at $\mathsf{Fin}\ n$, and is therefore contractible.

Alternatively, we can exploit additional principles in HoTT to consider operadic structures on collections of operations indexed over the groupoid of finite sets and bijections, i.e. finite sets that are not always totally-ordered. A common presentation of this groupoid, Bishop-finite sets [26, 24], internalises finiteness as a predicate on types.

▶ **Definition 4** (Bishop-finite sets). *A type $A : \mathit{Type}$ is* Bishop-finite *when*

$$\mathit{isFinite}\ A = \sum[\ n \in \mathbb{N}\ ]\ \|\ A \simeq \mathit{Fin}\ n\ \|$$

is inhabited, where $\| X \|$ denotes the propositional truncation of X. The universe of Bishop-finite sets is $\mathit{FinSet} = \sum[\ A \in \mathit{Type}\ ]\ \mathit{isFinite}\ A$, with interpretation map $\mathit{El} : \mathit{FinSet} \to \mathit{Type}$ given by the first projection.

For any type A, there is a well-known equivalence between the type $\mathsf{isFinite}\ A$ and the h-proposition $\| \sum[\ n \in \mathbb{N}\ ]\ A \simeq \mathsf{Fin}\ n \|$. Intuitively, this equivalence witnesses that every finite set has a unique cardinality, and therefore this notion of finiteness is an h-proposition.

Although our definition of FinSet is a large type, it can equivalently be defined as a small type by taking an appropriate groupoid quotient of the naturals. Accordingly, we will ignore size issues related to FinSet throughout this paper. Importantly, El is an embedding, which follows from the proof that the finiteness of a type is an h-proposition. That is, for any finite sets $A\ B : \mathsf{FinSet}$, the functorial action of El on paths is an equivalence between the path space of codes $A \equiv B$ and the path space of the underlying types $\mathsf{El}\ A \equiv \mathsf{El}\ B$. A Tarski universe with this property is also known as a univalent universe [11, 9, 10]. Since FinSet is a subtype of Type whose predicate $\mathsf{isFinite}$ is an h-proposition, FinSet is also a *univalent subuniverse* of the ambient universe: its codes inherit the path structure of the underlying types. We recall that the property that a type A is an h-set, i.e. $(x\ y : A)\ (p\ q : x \equiv y) \to p \equiv q$, is itself an h-proposition. Therefore, given $(A\ ,\ n\ ,\ p) : \mathsf{FinSet}$, it follows from the truncated equivalence between A and $\mathsf{Fin}\ n$ witnessed by p that, since $\mathsf{Fin}\ n$ is an h-set, so is A. That is, for every finite set $A : \mathsf{FinSet}$ the underlying type $\mathsf{El}\ A$ is an h-set. Therefore, given any two finite sets $A\ B : \mathsf{FinSet}$, by univalence the type of paths $A \equiv B$ is equivalent to the type of equivalences $\mathsf{El}\ A \simeq \mathsf{El}\ B$. We express the forward direction of this equivalence by:

$$\mathsf{un} : (A\ B : \mathsf{FinSet}) \to \mathsf{El}\ A \simeq \mathsf{El}\ B \to A \equiv B$$

The universe FinSet is closed under common type formers including dependent sums and products. In particular, for every finite set $A : \mathsf{FinSet}$ and family of finite sets $B : \mathsf{El}\ A \to \mathsf{FinSet}$, we have $\hat{\sum}\ A\ B : \mathsf{FinSet}$ and $\hat{\prod}\ A\ B : \mathsf{FinSet}$, together with definitional equalities $\mathsf{El}\ (\hat{\sum}\ A\ B) = \sum[\ a \in \mathsf{El}\ A\ ]\ \mathsf{El}\ (B\ a)$ and $\mathsf{El}\ (\hat{\prod}\ A\ B) = (a : \mathsf{El}\ A) \to \mathsf{El}\ (B\ a)$. Finiteness proofs for $\mathsf{isFinite}\ (\sum[\ a \in \mathsf{El}\ A\ ]\ \mathsf{El}\ (B\ a))$ and $\mathsf{isFinite}\ ((a : \mathsf{El}\ A) \to \mathsf{El}\ (B\ a))$ can be found in our Cubical Agda formalisation, and in the UniMath Coq library [24]. Our formalisation also shows that the universe of Bishop-finite sets is closed under isomorphism. That is, for any two finite sets $A\ B : \mathsf{FinSet}$ we can construct $A \mathrel{\hat{\equiv}} B : \mathsf{FinSet}$ with a definitional equality $\mathsf{El}\ (A \mathrel{\hat{\equiv}} B) = \mathsf{El}\ A \simeq \mathsf{El}\ B$ and a proof of $\mathsf{isFinite}\ (\mathsf{El}\ A \simeq \mathsf{El}\ B)$.

From this notion of Bishop-finite sets internalised in HoTT, we can define a collection of finitary operations as a FinSet-indexed family of h-sets. Indeed, such families are also known as *combinatorial species* [20, 26], or equivalently, *symmetric finitary containers* [16, 19]: finitary containers whose position type carries a symmetric-group action induced by the indexing groupoid FinSet. Every such family $K : \mathsf{FinSet} \to \mathsf{Type}$ comes equipped with a canonical right action of symmetric groups of the form $\mathsf{El}\ A \simeq \mathsf{El}\ A$, or equivalently $A \equiv A$, for $A : \mathsf{FinSet}$. Importantly, $\mathsf{El}\ A \simeq \mathsf{El}\ A$ and $A \equiv A$ are equivalent not just as sets, but also as groups. In particular, for every symmetric group element $\sigma : A \equiv A$, this action is given by the automorphism witnessed by $\mathsf{subst}\ K\ \sigma : K\ A \to K\ A$ and $\mathsf{subst}\ K\ (\mathsf{sym}\ \sigma) : K\ A \to K\ A$. We note that as K is a family of h-sets, the group of paths $K\ A \equiv K\ A$ is equivalent to the group of automorphisms $K\ A \simeq K\ A$. It follows from the laws of substitution over paths that this construction from $A \equiv A$ to $K\ A \equiv K\ A$ is a group homomorphism. That is, up to a path, substitution over the identity is equal to the identity and substitution over the composite of paths is equal to the composition of substitutions. The proof of the symmetry law follows from the identity and composition laws, and the symmetry law on paths: $p \cdot \mathsf{sym}\ p \equiv \mathsf{refl}$.

So far, we have seen how FinSet-indexed families have a canonical notion of right action of symmetric groups, in contrast to that of $\mathbb{N}$-indexed families. However, we haven't described what it means for a family of types indexed over FinSet to have an operadic structure. To do so, we begin by considering the definition of planar operads and replacing the finite summation function sum with the dependent sum type former of finite sets $\hat{\sum}$. The intuitive connection is that sum is the dependent sum type former for the Tarski universe with codes given by $\mathbb{N}$ and interpretation map $\mathsf{Fin} : \mathbb{N} \to \mathsf{Type}$, i.e. the universe of totally-ordered finite sets. For example, given a collection of finitary operations $K : \mathsf{FinSet} \to \mathsf{Type}$, a finite set $A : \mathsf{FinSet}$, and family of finite sets $B : \mathsf{El}\ A \to \mathsf{FinSet}$, the composition map of an operad $O : \mathsf{Operad}\ K$ is given by a field

$$\mathsf{comp}\ O\ A\ B : K\ A \to ((a : \mathsf{El}\ A) \to K\ (B\ a)) \to K\ (\hat{\sum}\ A\ B)$$

Moreover, given the singleton finite set $\top : \mathsf{FinSet}$ and a family of finite sets $C : (a : \mathsf{El}\ A) \to \mathsf{El}\ (B\ a) \to \mathsf{FinSet}$, the right identity and associativity laws are now given as heterogeneous paths over the following canonical paths:

$$\hat{\sum}\text{-}\mathsf{idr}\ A : \hat{\sum}\ A\ (\lambda\ a \to \top) \equiv A$$
$$\hat{\sum}\text{-}\mathsf{assoc}\ A\ B\ C : \hat{\sum}\ A\ (\lambda\ a \to \hat{\sum}\ (B\ a)\ (C a)) \equiv \hat{\sum}\ (\hat{\sum}\ A\ B)\ (\lambda\ (a\ ,\ b) \to C\ a\ b)$$

We can construct these paths by applying un to the corresponding isomorphisms in the meta-theory, i.e. for all $A : \mathsf{Type}$, $B : A \to \mathsf{Type}$ and $C : (a : A) \to B\ a \to \mathsf{Type}$, we construct $\hat{\sum}$-idr and $\hat{\sum}$-assoc by applying un to the following canonical isomorphisms:

$$\sum\text{-}\mathsf{idr} : A \times \top \simeq A$$
$$\sum\text{-}\mathsf{assoc} : \sum[\ a \in A\ ]\ \sum[\ b \in B\ a\ ]\ C\ a\ b \simeq \sum[\ (a\ ,\ b) \in \sum[\ a \in A\ ]\ B\ a\ ]\ C\ a\ b$$

Furthermore, given any operation $k : K\ A$ it is also now necessary to witness the left identity as a heterogeneous path as follows:

$$\mathsf{idl}\ O\ A\ k : \mathsf{PathP}\ (\lambda\ i \to K\ (\hat{\sum}\text{-}\mathsf{idl}\ A\ i))\ (\mathsf{comp}\ O\ A\ (\lambda\ a \to \top)\ k\ (\lambda\ a \to \mathsf{id}\ O))\ k,$$

where the path $\hat{\sum}\text{-}\mathsf{idl}\ A : \hat{\sum}\ \top\ (\lambda\ t \to A) \equiv A$ is constructed by the application of $\mathsf{un}\ (\hat{\sum}\ \top\ (\lambda\ t \to A))\ A$ to the canonical isomorphism that exists between $\top \times \mathsf{El}\ A$ and $\mathsf{El}\ A$.

To complete our definition of a symmetric operad, we might consider adding a field witnessing that our previously described right actions of symmetric groups on the collection of operations K is compatible with the composition map. That is, we might introduce an analogue to the symm field for FinSet-indexed families. In particular, for every finite set $A : \mathsf{FinSet}$, family of finite sets $B : \mathsf{El}\ A \to \mathsf{FinSet}$, operation $k : K\ A$, family of operations $ks : (a : \mathsf{El}\ A) \to K\ (\mathsf{El}\ (B\ a))$ and symmetric group element $\sigma : A \equiv A$, we could introduce the following path as a field:

```
symm O A B k ks σ : let p i a = transp (λ j → El (σ (i ∨ j))) i a in
  PathP (λ i → K (Σ̂ (σ i) (λ a → B (p i a))))
        (comp O A (B ∘ subst El σ) k (ks ∘ subst El σ))
        (comp O A B (subst K σ k) ks)
```

However, it turns out that this form of symm, where the right action of a symmetric group is defined by path substitution, can already be constructed in Cubical Agda as follows:

```
symm O A B k ks σ i =
  comp O (σ i) (λ a → B (transp (λ j → El (σ (i ∨ j))) i a))
                 (transp (λ j → K (σ (i ∧ j))) (∼ i) k)
                 (λ a → ks (transp (λ j → El (σ (i ∨ j))) i a))
```

That is, by switching from $\mathbb{N}$ to FinSet indexed families, and changing from sum to the dependent sum type former $\hat{\Sigma}$, our notion of an operad restricts to the classical definition which includes compatability with the right actions of symmetric groups.

▶ **Definition 5** (Symmetric operad on a FinSet-indexed family)**.** *Given a family of h-sets* $K : \mathit{FinSet} \to \mathit{hSet}$*, a* symmetric operad *on* K *is a term of the record type* $\mathit{Operad}\ K$ *whose fields are obtained from Definition 1 by replacing* sum *by* $\hat{\Sigma}$*,* $\mathit{Fin}\ n$ *by* $\mathit{El}\ A$*, and the indexing* $\mathbb{N}$ *by* FinSet*. The record retains the same five fields* id*,* comp*,* idl*,* idr *and* assoc*; the equivariance law is derivable from these fields together with the substitution actions of* FinSet*, and so does not appear as a separate field.*

We can similarly translate our notion of planar operad morphism to use FinSet and $\hat{\Sigma}$, and we don't require a field witnessing preservation of the actions of symmetric groups. In particular, for any two families $K\ L : \mathsf{FinSet} \to \mathsf{Type}$, family of maps $(A : \mathsf{FinSet}) \to K\ A \to L\ A$, operation $k : K\ A$, and symmetric group element $\sigma : A \equiv A$, a proof of $f\ A\ (\mathsf{subst}\ K\ \sigma\ k) \equiv \mathsf{subst}\ L\ \sigma\ (f\ A\ k)$ follows directly from substitution commuting with morphisms in slice categories.

▶ **Example 6** (Symmetric expression operad SymExpr)**.** We recall the example of the planar operadic structure on the type family IExpr : $\mathbb{N} \to$ Type, and introduce a near-identical FinSet-indexed analogue.

```
data SymExpr : FinSet → Type₁ where
  id↑ : SymExpr ⊤
  val↑ : ℕ → SymExpr ⊥
  add↑ : SymExpr A → SymExpr B → SymExpr (A ⊎ B)
```

In this example, the type former _⊎_ : FinSet → FinSet → FinSet corresponds to coproducts in the universe of finite sets. That is, $\mathsf{El}\ (A \uplus B) = \mathsf{El}\ A \uplus \mathsf{El}\ B$, and $\bot$ is the empty type.

The proof that finite sets are closed under finite coproducts can again be found both in our Cubical Agda formalisation and in the UniMath library [24]. Notably, SymExpr is a family of large types because it is indexed over FinSet and add↑ is parameterised over a large type. In Section 6 we address this with a small type of unordered finite sets.

To define the operad composition map for the family SymExpr, for all types $A\ B$: Type and families $C : A \uplus B \to$ Type, we will make use of the following canonical isomorphism:

$$\begin{aligned} \uplus\text{-distr}\ A\ B\ C : (\textstyle\sum[\ a \in A\ ]\ C\ (\mathsf{inl}\ a)) \uplus (\sum[\ b \in B\ ]\ C\ (\mathsf{inr}\ b)) \\ \simeq (\textstyle\sum[\ x \in A \uplus B\ ]\ C\ x) \end{aligned}$$

where $\uplus$-distr $A\ B\ C$ is constructed in the evident way. Moreover, for all families $A : \top \to$ Type and $B : \bot \to$ Type, we will use the following two canonical isomorphisms:

$$\textstyle\sum\text{-idl}\ A : A\ \mathsf{tt} \simeq \sum[\ x \in \top\ ]\ A\ x \qquad\qquad \sum\text{-}\bot\ B : \bot \simeq \sum[\ x \in \bot\ ]\ B\ x$$

We can then construct the composition map for an operad O : Operad SymExpr as follows:

```
comp O .⊤ B id↑ es = subst SymExpr (un (El  (B tt)) (∑[ x ∈ ⊤ ] B x) (∑-idl B)) (es tt)
comp O .⊥ B (val↑ n) es = subst SymExpr (un ⊥ (∑[ x ∈ ⊥ ] B x) (∑-⊥ B)) (val↑ n)
comp O .(A₁ ⊎ A₂) B (add↑ e₁ e₂) es =
  let es₁ = comp O A₁ (B ∘ inl) e₁ (es ∘ inl)
      es₂ = comp O A₂ (B ∘ inr) e₂ (es ∘ inr)
   in subst SymExpr (un _ _ (⊎-distr A₁ A₂ B)) (add↑ es₁ es₂)
```

Proofs that this construction respects the necessary identity and associativity laws are given in our Cubical Agda formalisation.

6 Generalised Operad Universes

We have seen how symmetric and planar operads can be defined in HoTT by varying the Tarski universe that indexes operations. For the planar case, we considered operations indexed by the universe of totally-ordered finite sets $(\mathbb{N}, \mathsf{Fin})$, while for the symmetric case we considered the more general groupoid of finite sets and bijections $(\mathsf{FinSet}, \mathsf{El})$. To describe how an operadic composition map acts on indices, we required both universes to be closed under dependent sums and have a unit element. In particular, for the planar case we required the finite summation map sum together with the unit 1, while for the symmetric case we used closure of finite sets under dependent sums $\hat{\sum}$ together with the singleton finite set $\top$. As the symmetric and planar operad definitions are otherwise uniform, this suggests a generalisation for collections of operations indexed by any universe with sufficient closure properties. In this section, we introduce an internal notion of such universes in HoTT, which allows us to work with a generalised notion of operads that supports many important operadic constructions being defined without the need to separate details for the symmetric and planar versions.

We begin by recalling that a Tarski type universe comprises a type of codes U : Type and an interpretation family $E : U \to$ Type. For example, the groupoid of finite sets and bijections forms a universe with $U =$ FinSet and $E =$ El. In Agda, such universes can be defined as a record type $\mathcal{U}$: Universe with a field for the type of codes Code $\mathcal{U}$: Type, and for every code A : Code $\mathcal{U}$ a field $[\![\ \mathcal{U} \ni A\ ]\!]$: Type corresponding to the underlying type. To ensure universes are closed under dependent sums, we first introduce the requirement that $\mathcal{U}$ must come equipped with the type former

$$\hat{\textstyle\sum} : (A : \mathsf{Code}\ \mathcal{U}) \to ([\![\ \mathcal{U} \ni A\ ]\!] \to \mathsf{Code}\ \mathcal{U}) \to \mathsf{Code}\ \mathcal{U}$$

such that for every $A : \mathsf{Code}\ \mathcal{U}$ and $B : [\![\ \mathcal{U} \ni A\ ]\!] \to \mathsf{Code}\ \mathcal{U}$, we have an equivalence:

$$[\![\hat{\sum}]\!]\ A\ B : [\![\ \mathcal{U} \ni (\hat{\sum}\ A\ B)\ ]\!] \simeq \sum[\ a \in [\![\ \mathcal{U} \ni A\ ]\!]\ ]\ [\![\ \mathcal{U} \ni B\ a\ ]\!].$$

In addition to closure under dependent sums, we require a field $\hat{\top} : \mathsf{Code}\ \mathcal{U}$ corresponding to the code for the unit type, together with an equivalence $[\![\ \mathcal{U} \ni \hat{\top}\ ]\!] \simeq \top$. However, these properties are not sufficient to show $\mathcal{U}$ is closed under dependent sums and a unit object. For example, for every $A : \mathsf{Code}\ \mathcal{U}$, $B : [\![\ \mathcal{U} \ni A\ ]\!] \to \mathsf{Code}\ \mathcal{U}$ and $C : (a : [\![\ \mathcal{U} \ni A\ ]\!]) \to [\![\ \mathcal{U} \ni B\ a\ ]\!] \to \mathsf{Code}\ \mathcal{U}$, these fields are insufficient to construct a proof of associativity:

$$\hat{\sum}\text{-assoc}\ A\ B\ C : \hat{\sum}\ A\ (\lambda\ a \to \hat{\sum}\ (B\ a)\ (C\ a)) \equiv \hat{\sum}\ (\hat{\sum}\ A\ B)\ (\mathsf{uncurry}\ C \circ [\![\hat{\sum}]\!]\ A\ B)$$

To address this, it suffices to require a map from the path space between any two underlying types in a universe to the path space between their codes. Concretely, this means that for any two codes $A\ B : \mathsf{Code}\ \mathcal{U}$, we require the following extra field:

$$\mathsf{Inj}\ A\ B : [\![\ \mathcal{U} \ni A\ ]\!] \simeq [\![\ \mathcal{U} \ni B\ ]\!] \to A \equiv B$$

However, the Inj field alone is insufficient to map the higher path structure between underlying types in a universe to the corresponding higher path structure between their codes. In particular, we recall that this notion of a generalised operad universe will be used to index a family of h-sets corresponding to the operations of an operad. That is, given a universe $\mathcal{U} : \mathsf{Universe}$, a $\mathcal{U}$-operad has operations given by a family of h-sets $K : \mathsf{Code}\ \mathcal{U} \to \mathsf{Type}$. Importantly, the type of h-sets is an h-groupoid, and hence the higher path structure of codes beyond that of the 1-groupoid structure, i.e. paths between paths, is trivially respected by K.

Hence, it is sufficient to include a coherency condition witnessing that Inj preserves this 1-groupoid structure. For all codes $A\ B\ C : \mathsf{Code}\ \mathcal{U}$ and equivalences $e_1 : [\![\ \mathcal{U} \ni A\ ]\!] \simeq [\![\ \mathcal{U} \ni B\ ]\!]$ and $e_2 : [\![\ \mathcal{U} \ni B\ ]\!] \simeq [\![\ \mathcal{U} \ni C\ ]\!]$, we can achieve this by adding the field:

$$\mathsf{InjComp}\ A\ B\ C\ e_1\ e_2 : \mathsf{Inj}\ A\ C\ (e_1 \cdot e_2) \equiv \mathsf{Inj}\ A\ B\ e_1 \cdot \mathsf{Inj}\ B\ C\ e_2$$

Intuitively, InjComp asserts that Inj must preserve composition of equivalences, mapping composition of equivalences to the composition of paths. Moreover, this condition is sufficient to witness that up to a path Inj maps identity equivalences to reflection paths.

▶ **Proposition 7** (InjRefl). *For every code $A : \mathsf{Code}\ \mathcal{U}$ there is a path*

$$\mathsf{InjRefl}\ A : \mathsf{Inj}\ A\ A\ (\mathsf{id}\simeq\ [\![\ \mathcal{U} \ni A\ ]\!]) \equiv \mathsf{refl},$$

where $\mathsf{id}\simeq\ [\![\ \mathcal{U} \ni A\ ]\!] : [\![\ \mathcal{U} \ni A\ ]\!] \simeq [\![\ \mathcal{U} \ni A\ ]\!]$ is the identity equivalence on $[\![\ \mathcal{U} \ni A\ ]\!]$.

Proof. To construct the path InjRefl, we first apply InjComp $A\ A\ A$ to the identity equivalence (twice) on $[\![\ \mathcal{U} \ni A\ ]\!]$ and precompose with the functorial action of Inj $A\ A$ on paths applied to the path witnessing that the identity equivalence composed with itself is the identity equivalence. This construction gives us a path between Inj $A\ A$ (id$\simeq$ $[\![\ \mathcal{U} \ni A\ ]\!]$) and the same path composed with itself. It follows from the groupoid structure of paths that for any type A, term $x : A$, and path $p : x \equiv x$, if there is a path of type $p \cdot p \equiv p$ then we can construct a path of type $p \equiv \mathsf{refl}$. We note that this is a purely path-algebraic fact about idempotent loops, obtained by left-cancelling p against itself, and does not require A to be a set. Hence, by applying this rule we can construct a path between Inj $A\ A$ (id$\simeq$ $[\![\ \mathcal{U} \ni A\ ]\!]$) and refl. ◀

▶ **Proposition 8** (InjSec). *For all codes $A\ B : \mathsf{Code}\ \mathcal{U}$ and paths $p : A \equiv B$, there is a path*

$$\mathsf{InjSec}\ A\ B\ p : \mathsf{Inj}\ A\ B\ (\mathsf{pathToEquiv}\ (\mathsf{cong}\ [\![\ \mathcal{U} \ni \_]\!]\ p)) \equiv p.$$

Proof. For any types $X\ Y$: Type, if we let pathToEquiv $X\ Y : X \equiv Y \to X \simeq Y$ be the canonical map from paths to equivalences then by applying the J-elimination rule to the composition of the path witnessing that pathToEquiv maps reflection to the identity equivalence and the path InjRefl A, we can construct the above family of paths. ◀

In particular, InjSec witnesses that Inj has all sections given by the composition of the functorial action of the interpretation map $[\![\ \mathcal{U} \ni \_]\!]$ on paths and the canonical map from paths to equivalences. Intuitively, this condition asserts that the path spaces of the representing codes of a universe precisely reflect the path spaces of the underlying types, and no more.

To conclude our description of generalised operad universes, we first recall that an operadic structure on a collection of operations includes heterogeneous paths that witness that the composition map respects identity and associativity laws. In particular, for generalised operad universes these paths will necessarily be heterogeneous over Inj applied to the following canonical equivalences, which correspond to left/right identity and associativity,

$$\begin{aligned}
\hat{\textstyle\sum}\mathsf{Idl}{\simeq}\ \mathcal{U}\ A &: [\![\ \mathcal{U} \ni \hat{\textstyle\sum}\ \hat{\top}\ (\lambda\ t \to A)\ ]\!] \simeq [\![\ \mathcal{U} \ni A\ ]\!] \\
\hat{\textstyle\sum}\mathsf{Idr}{\simeq}\ \mathcal{U}\ A &: [\![\ \mathcal{U} \ni \hat{\textstyle\sum}\ A\ (\lambda\ a \to \hat{\top})\ ]\!] \simeq [\![\ \mathcal{U} \ni A\ ]\!] \\
\hat{\textstyle\sum}\mathsf{Assoc}{\simeq}\ \mathcal{U}\ A\ B\ C &: [\![\ \mathcal{U} \ni \hat{\textstyle\sum}\ A\ (\lambda\ a \to \hat{\textstyle\sum}\ (B\ a)\ (C\ a))\ ]\!] \simeq \\
&\qquad [\![\ \mathcal{U} \ni \hat{\textstyle\sum}\ (\hat{\textstyle\sum}\ A\ B)\ (\mathsf{uncurry}\ C \circ \mathsf{fun}\ ([\![\hat{\textstyle\sum}]\!]\ A\ B))\ ]\!]
\end{aligned}$$

for all A : Code $\mathcal{U}$, $B : [\![\ \mathcal{U} \ni A\ ]\!] \to$ Code $\mathcal{U}$ and $C : (a : [\![\ \mathcal{U} \ni A\ ]\!]) \to [\![\ \mathcal{U} \ni B\ a\ ]\!] \to$ Code $\mathcal{U}$. These are precisely the equivalences constructed from the underlying canonical equivalences on the dependent sum construction in the meta-theory composed with equivalences constructed from application of $[\![\hat{\textstyle\sum}]\!]$. To construct $\mathcal{U}$-operadic structures, we require each of the above equivalences to be represented by the paths they are sent to by Inj. That is, we require paths between the above equivalences and the application of the map that sends each $X\ Y$: Code $\mathcal{U}$ and $e : [\![\ \mathcal{U} \ni A\ ]\!] \simeq [\![\ \mathcal{U} \ni A\ ]\!]$ to pathToEquiv (cong $[\![\ \mathcal{U} \ni \_]\!]$ (Inj $X\ Y\ e$)) to the same equivalences. Equivalently, given the inverse map to pathToEquiv following from univalence, i.e. ua : $X \simeq Y \to X \equiv Y$, we can capture the required conditions with the following fields:

$$\begin{aligned}
&[\![\hat{\textstyle\sum}\mathsf{Idl}]\!]\ \mathcal{U}\ A : \mathsf{ua}\ (\hat{\textstyle\sum}\mathsf{Idl}{\simeq}\ \mathcal{U}\ A) \equiv \mathsf{cong}\ [\![\ \mathcal{U} \ni _]\!]\ (\mathsf{Inj}\ _\ _\ (\hat{\textstyle\sum}\mathsf{Idl}{\simeq}\ \mathcal{U}\ A)) \\
&[\![\hat{\textstyle\sum}\mathsf{Idr}]\!]\ \mathcal{U}\ A : \mathsf{ua}\ (\hat{\textstyle\sum}\mathsf{Idr}{\simeq}\ \mathcal{U}\ A) \equiv \mathsf{cong}\ [\![\ \mathcal{U} \ni _]\!]\ (\mathsf{Inj}\ _\ _\ (\hat{\textstyle\sum}\mathsf{Idr}{\simeq}\ \mathcal{U}\ A)) \\
&[\![\hat{\textstyle\sum}\mathsf{Assoc}]\!]\ \mathcal{U}\ A\ B\ C : \mathsf{ua}\ (\hat{\textstyle\sum}\mathsf{Assoc}{\simeq}\ \mathcal{U}\ A\ B\ C) \equiv\ \mathsf{cong}\ [\![\ \mathcal{U} \ni _]\!]\ (\mathsf{Inj}\ _\ _\ (\hat{\textstyle\sum}\mathsf{Assoc}{\simeq}\ \mathcal{U}\ A\ B\ C))
\end{aligned}$$

Collecting the fields together, we obtain the following record type.

▶ **Definition 9** (Generalised operad universe). *A* generalised operad universe *is a record* $\mathcal{U}$: *Universe comprising a type of codes* Code $\mathcal{U}$ *and an interpretation family* $[\![\ \mathcal{U} \ni \_\ ]\!]$*; a code* $\hat{\top}$ *for the unit together with an equivalence* $[\![\hat{\top}]\!]$*; a code former* $\hat{\textstyle\sum}$ *for dependent sums together with an equivalence* $[\![\hat{\textstyle\sum}]\!]$*; a representation map* Inj *satisfying the* 1*-coherence* InjComp*; and three path-level closure laws* $[\![\hat{\textstyle\sum}\mathit{Idl}]\!]$*,* $[\![\hat{\textstyle\sum}\mathit{Idr}]\!]$ *and* $[\![\hat{\textstyle\sum}\mathit{Assoc}]\!]$*, as described above.*

Both the universe of totally-ordered finite sets and the groupoid of finite sets and bijections satisfy all the above criteria. Another example of a universe for which all the above terms can be constructed, is the universe Type itself. Similarly, for any homotopy-level $n \geq -1$, i.e. propositions or higher, the universe of n-types also satisfies these criteria.

We note that our construction does not require that the type of codes Code $\mathcal{U}$ is a 1-type. Since the operations of an operad will form a family K : Code $\mathcal{U} \to$ hSet, and hSet is itself a 1-type, every such family factors through the groupoid truncation $\|$ Code $\mathcal{U}\ \|_1$. Equivalently, families of h-sets over Code $\mathcal{U}$ are equivalent to families of h-sets over its 1-truncation, so our

theory of operads never inspects any higher path structure of Code $\mathcal{U}$ beyond its 1-groupoid truncation. Accordingly, it suffices to require that Inj preserves 1-groupoid structure as witnessed by InjComp, together with a proof of closure under dependent sums and a unit type, alongside the 1-coherences $[\![\hat{\sum}\mathsf{Idl}]\!]$, $[\![\hat{\sum}\mathsf{Idr}]\!]$ and $[\![\hat{\sum}\mathsf{Assoc}]\!]$. The resulting notion of a generalised operad universe is closely related to the univalent universes of Christensen [11], and in particular the univalent universe of finite types studied by Carette and Sabry [9] and Choudhury, Karwowski and Sabry [10].

From this characterisation of universes closed under both dependent sums and a unit type, our internalisation of operads can be generalised by parametrising over these universes.

▶ **Definition 10** ($\mathcal{U}$-operad). *Given a generalised operad universe $\mathcal{U}$ and a family of h-sets K : Code $\mathcal{U}$ → hSet, a $\mathcal{U}$-operad on K is a term of a parameterised record type Operad $\mathcal{U}$ K : Type whose fields are obtained from Definition 1 by replacing finite summation sum and dependent sum over finite sets by the universe-supplied $\hat{\sum}$, and replacing the indexing $\mathbb{N}$ or FinSet by Code $\mathcal{U}$.*

Given such an operadic structure O : Operad $\mathcal{U}$ K, its fields are almost identical to those detailed in Sections 4 and 5, with finite summation or dependent sum over finite sets replaced by $\hat{\sum}$. For example, for every A : Code $\mathcal{U}$ and B : $[\![\ \mathcal{U} \ni A\ ]\!] \to$ Code $\mathcal{U}$ the composition map for the operad O is given by the following field:

$$\mathsf{comp}\ O\ A\ B : K\ A \to ((a : [\![\ \mathcal{U} \ni A\ ]\!]) \to K\ (B\ a)) \to K\ (\hat{\sum}\ A\ B)$$

Similarly, the unit operation is given by a field $\mathsf{id}\ O : K\ \hat{\top}$. The families of heterogeneous paths witnessing the left/right unit laws and the associativity law can be defined by appropriately applying Inj to the respective laws on dependent sums. For example, for every A : Code $\mathcal{U}$ and $k : K\ A$ the corresponding left unit law for O is witnessed by the following field:

$$\begin{aligned}&\mathsf{idl}\ O\ A\ k : \mathsf{PathP}\\ &\quad(\lambda\ i \to K\ (\mathsf{Inj}\ (\hat{\sum}\ \hat{\top}\ (\lambda\ u \to A))\ A\ (\textstyle\sum\mathsf{Idl}\ [\![\ \mathcal{U} \ni A\ ]\!])\ i))\\ &\quad(\mathsf{comp}\ O\ \hat{\top}\ (\lambda\ u \to A)\ (\mathsf{id}\ O)\ (\lambda\ u \to k))\ k\end{aligned}$$

To conclude our discussion on generalised operads, we describe the groupoid structure on $\mathcal{U}$-operads, which arises from the path space on the type of $\mathcal{U}$-operads. In HoTT, given any type A and family of h-propositions $P : A \to \mathsf{Type}$, the sum type $\sum[\ a \in A\ ]\ P\ a$ can be considered a *subtype* of A, as witnessed by the first projection $\pi_1 : \sum[\ a \in A\ ]\ P\ a \to A$ being an embedding. Concretely, this means that the functorial action of the first projection map on paths, i.e. $\mathsf{cong}\ \pi_1 : x \equiv y \to \pi_1\ x \equiv \pi_1\ y$, is an equivalence. As such, the path space of a subtype is characterised by the path space of its underlying type. Given a collection of operations K : Code $\mathcal{U}$ → Type, where K is a family of h-sets and is equipped with an operad structure O : Operad $\mathcal{U}$ K, the types of each of the paths $\mathsf{idl}\ O\ n\ k$, $\mathsf{idr}\ O\ n\ k$ and $\mathsf{assoc}\ O\ n\ ns\ nss\ k\ ks\ kss$ are then h-propositions. Hence, an operad can be understood as a subtype of a record type with only an identity id and composition map comp. That is, the operad laws are not important in characterising the path spaces between operads, and we make use of this fact to simplify the following definition of these path spaces.

Given any two collections of operations $K\ L$: Code $\mathcal{U}$ → Type, where K and L are families of h-sets with operad structures O^K : Operad $\mathcal{U}$ K and O^L : Operad $\mathcal{U}$ L, and given any family of paths $K{\equiv}L : (A : \mathsf{Code}\ \mathcal{U}) \to K\ A \equiv L\ A$, the heterogeneous path space between O^K and O^L is equivalent to a pair of heterogeneous paths ranging over $K{\equiv}L$ between the identities and composition maps of O^K and O^L. Importantly, it follows from K and L being families of h-sets that this pair is an h-proposition.

▶ **Proposition 11** (Groupoid structure on the type of $\mathcal{U}$-operads). *For any universe $\mathcal{U}$ and any two $\mathcal{U}$-operads (K, O^K) and (L, O^L), the heterogeneous path space between O^K and O^L over a family of paths $K \equiv L$ is an h-proposition. Consequently, the type* $\sum[\ K \in (\mathsf{Code}\ \mathcal{U} \to \mathsf{hSet})\ ]\ \mathsf{Operad}\ \mathcal{U}\ K$ *of all $\mathcal{U}$-operads carries a 1-groupoid structure induced by the group structure on path spaces.*

We conclude with some remarks on the utility of our generalised operad construction. In particular, by developing a generalised formulation of operads, we can often introduce further operadic ideas by parameterising over a generalised operad universe. This allows us to define concepts such as operad morphisms, as well as algebras and monads over an operad, each of which are uniform over the universe, without specialising for the planar and symmetric cases. Indeed, as we show in Section 9, even the free operad construction can be parameterised over the universe of codes. This highlights one of the key benefits of studying operads in HoTT, namely that the functoriality of type families with respect to paths enables a generalised notion of an operad that supports uniform definitions for many constructions. For example, as shown in our formalisation, the *associative*, *commutative* and endomorphism operads can all be defined uniformly over generalised operad universes, such that specialising to totally ordered finite sets or Bishop-finite sets gives the expected planar and symmetric versions.

7 Category of Operads

To work with operads in HoTT, we require both structure-preserving maps between them and concrete interpretations of the abstract collections of operations that they represent. These correspond, respectively, to operad homomorphisms and operad algebras. A concrete example of both is the fill function from Section 4, which can be understood as a morphism from the PartialList operad to the *endomorphism operad* on lists. In this section, we begin by presenting the categorical structure of operads within HoTT. We then introduce the endomorphism operad on a given type, whereby morphisms in the category of operads into the endomorphism operad constitute the notion of an operad algebra.

In addition to the groupoid structure described in the previous section, the (large) type $\sum[\ K \in (\mathsf{Code}\ \mathcal{U} \to \mathsf{hSet})\ ]\ \mathsf{Operad}\ \mathcal{U}\ K$ has a more general category structure where a morphism between operads is any family of maps between their operations that preserves composition and identity.

▶ **Definition 12** (Operad morphism and the category of $\mathcal{U}$-operads). *Given operads (K, O^K) and (L, O^L), an* operad morphism *$f : O^K \Rightarrow O^L$ is a term of a record type with a projection $\langle\ f\ \rangle : (A : \mathsf{Code}\ \mathcal{U}) \to K\ A \to L\ A$ and proofs that f respects operad identity and composition. The category of $\mathcal{U}$-operads has $\mathcal{U}$-operads as objects and operad morphisms as arrows.*

As with the path space between operads, this type is a subtype of the type of slice morphisms $(A : \mathsf{Code}\ \mathcal{U}) \to K\ A \to L\ A$, with the identity and composition laws as h-propositional fields. This yields the following characterisation of equality.

▶ **Proposition 13** (Equality of operad morphisms). *For any two operad morphisms $f\ g : O^K \Rightarrow O^L$, the function $\mathsf{cong}\ \langle \_ \rangle : f \equiv g \to \langle\ f\ \rangle \equiv \langle\ g\ \rangle$ is an equivalence. That is, operad morphisms are equal if they agree on operations.*

The composition of operad morphisms is given by composing their underlying slice morphisms. In particular, given a third operad (J, O^J) and operad morphisms $f : O^J \Rightarrow O^K$ and $g : O^K \Rightarrow O^L$ we can construct the operations of their composition $\langle\ g \bullet f\ \rangle : O^J \Rightarrow O^L$ as $\langle\ g \bullet f\ \rangle\ A\ k = \langle\ g\ \rangle\ A\ (\langle\ f\ \rangle\ A\ k)$. The composition of slice morphisms respects both

operad identity and composition structure. The identity $\mathsf{id}^{op}\ O : O \Rightarrow O$ on $O : \mathsf{Operad}\ \mathcal{U}\ K$ is defined by $\langle\ O\ \rangle\ A\ k = k$, with both the proof of identity and composition preservation holding definitionally. Associativity and unitality of operad morphism composition then follow from the corresponding laws on slice morphisms via Proposition 13.

We have already seen two examples of planar operad morphisms in Section 4, namely $[\![\_]\!] : \mathsf{IExpr}\ n \to (\mathsf{Fin}\ n \to \mathsf{Expr}) \to \mathsf{Expr}$ and $\mathsf{fill} : \mathsf{PartialList}\ A\ n \to (\mathsf{Fin}\ n \to \mathsf{List}\ A) \to \mathsf{List}\ A$. Both are operad morphisms into an *endomorphism operad*, an operad whose operations are "n-ary like" functions on an h-set and whose composition map is function composition.

▶ **Definition 14** (Endomorphism operad $\mathsf{Endo}\ \mathcal{U}\ X$)**.** *For any h-set $X : \mathit{Type}$, the collection of operations $\mathit{EndoOps} : \mathit{Code}\ \mathcal{U} \to \mathit{Type}$ given by $\mathit{EndoOps}\ A = (\llbracket\ \mathcal{U} \ni A\ \rrbracket \to X) \to X$ forms a $\mathcal{U}$-operad structure $\mathit{Endo}\ \mathcal{U}\ X : \mathit{Operad}\ \mathcal{U}\ \mathit{EndoOps}$. We call this the* endomorphism $\mathcal{U}$-operad *on X and operad composition is simply given by function composition.*

Notably, when specialising to the universe of totally-ordered finite sets, EndoOps corresponds to the family of n-ary functions on X. Given that X is an h-set, EndoOps is also a family of h-sets. The unit operation $\mathsf{id}\ (\mathsf{Endo}\ \mathcal{U}\ X) : (\llbracket\ \mathcal{U} \ni \hat{\top}\ \rrbracket \to X) \to X$ is given by:

$$\mathsf{id}\ (\mathsf{Endo}\ \mathcal{U}\ X)\ f = f\ (\mathsf{inv}\ (\llbracket \hat{\top} \rrbracket\ \mathcal{U})\ \mathsf{tt})$$

Intuitively, the unit operation extracts the only element from a unitary collection $f : \llbracket\ \mathcal{U} \ni \hat{\top}\ \rrbracket \to X$. For every $A : \mathsf{Code}\ \mathcal{U}$, $B : \llbracket\ \mathcal{U} \ni A\ \rrbracket \to \mathsf{Code}\ \mathcal{U}$ together with an output operation $f : (\llbracket\ \mathcal{U} \ni A\ \rrbracket \to X) \to X$ and input operations $fs : (a : \llbracket\ \mathcal{U} \ni A\ \rrbracket) \to (\llbracket\ \mathcal{U} \ni B\ a\ \rrbracket \to X) \to X$, we define the composition map of the endomorphism operad on X as follows:

$$\mathsf{comp}\ (\mathsf{Endo}\ \mathcal{U}\ X)\ A\ B\ f\ fs\ xs = f\ (\lambda\ a \to fs\ a\ (\lambda\ b \to xs\ (\mathsf{inv}\ (\llbracket \hat{\sum} \rrbracket\ \mathcal{U}\ A\ B)\ (a\ ,\ b))))$$

To construct proofs of the identity and associativity laws, we first explain how the univalence map from equivalences to paths, i.e. $\mathsf{ua} : X \simeq Y \to X \equiv Y$, acts when appearing twice to the left of the function arrow. Concretely, for any types $X_1\ X_2\ Y\ Z : \mathsf{Type}$, equivalence $e : X_1 \simeq X_2$ and function $f : (X_1 \to Y) \to Z$, the action of $\mathsf{ua}\ e : X_1 \equiv X_2$ on f is witnessed by a heterogeneous path,

$$\mathsf{ua}{\to}{\to}\ f\ e : \mathsf{PathP}\ (\lambda\ i \to (\mathsf{ua}\ e\ i \to Y) \to Z)\ f\ (\lambda\ ys \to f\ (ys \circ \mathsf{fun}\ e))$$
$$\mathsf{ua}{\to}{\to}\ f\ e\ i\ ys = f\ (ys \circ \mathsf{ua\text{-}gluePt}\ e\ i)$$

where $\mathsf{ua\text{-}gluePt}\ e\ i : X_1 \to \mathsf{ua}\ e\ i$ is a path from the identity function to $\mathsf{fun}\ e$ and is a standard result. While we do not give the explicit constructions here, the proofs that the composition map of the endomorphism operad respects the identity and associativity laws proceed by application of $\mathsf{ua}{\to}{\to}$ to the equivalences $\hat{\sum}\mathsf{Idl}{\simeq}\ A$, $\hat{\sum}\mathsf{Idr}{\simeq}\ A$ and $\hat{\sum}\mathsf{Assoc}{\simeq}\ A\ B\ C$ followed by substitution along the paths $\llbracket \hat{\sum}\mathsf{Idl} \rrbracket$, $\llbracket \hat{\sum}\mathsf{Idr} \rrbracket$ and $\llbracket \hat{\sum}\mathsf{Assoc} \rrbracket$ respectively.

8 Monad over an Operad

When using operadic collections of operations, it is often useful to dependently build operations from data. Indeed, this is the behaviour of the canonical monad that arises over any operad. In particular, an element of this monad is an operation together with data stored at each input. Concretely, for any universe $\mathcal{U} : \mathsf{Universe}$ and collection of operations $K : \mathsf{Code}\ \mathcal{U} \to \mathsf{Type}$, where K is a family of h-sets, the monad over any $\mathcal{U}$-operad can be internalised in Cubical Agda by first introducing the following record type.

▶ **Definition 15** (The carrier OpM O X). *For an operad* O : *Operad* $\mathcal{U}$ K *and a type* X : *Type, the type* *OpM* O X *has terms given by the following record type:*

```
record OpM (O : Operad 𝒰 K) (X : Type) : Type where
  constructor _▷_▷_
  field
    Index : Code 𝒰
    Op : K Index
    Data : ⟦ 𝒰 ∋ Index ⟧ → X
```

When the type of codes for the universe $\mathcal{U}$ is an h-groupoid, then for any operad O : Operad $\mathcal{U}$ K the family OpM O restricts to an endofunctor on the category of h-sets and gives rise to a monad on h-sets in the usual sense, with the monadic laws holding as paths. Restricted to h-groupoids, the same construction extends to a 2-monad on the 2-category of h-groupoids, functions and paths, whose 2-cell coherence is provided in our Cubical Agda formalisation. Throughout the rest of this section, we refer to either structure simply as the monad over the operad O. The unit map for this monad corresponds to storing an element at the output of the unit operation, and can be defined as follows:

```
return : X → OpM O X
return x = ↑ 𝒰 ▷ unit O ▷ λ t → x
```

The composition map for this monad describes how given an output operation whose leaf data corresponds to input operations with their own data, we can use the operadic composition map to compose the output and input operations while retaining the data attached to the inputs. We define a dependent variation of this map whereby for every term ox : OpM O X and function ($\llbracket$ $\mathcal{U}$ ∋ Index ox $\rrbracket$ → X → OpM O Y), we construct a term compM ox f : OpM O Y as follows, where next ox f = λ i → f i (Data ox i):

```
Index (compM ox f) = ∑̂ 𝒰 (Index ∘ next ox f)
Op (compM ox f) = comp O (Index ox) (Index ∘ next ox f) (Op ox) (Op ∘ next ox f)
Data (compM ox f) k
  = let (i , j) = fun (⟦∑̂⟧ 𝒰 (Index ox) (Index ∘ next ox f)) k
    in Data (f i (Data ox i)) j
```

From this definition of compM, we can define the usual monad multiplication map, join : OpM O (OpM O X) → OpM O X, by simply mapping each o : OpM O (OpM O X) to compM o (λ i x → x). The functorial map _<$>_ : ($X$ → Y) → OpM O X → OpM O Y can also be defined using compM, however, this results in an index of $\hat{\sum}$ $\mathcal{U}$ (Index o) (λ i → ↑ $\mathcal{U}$), while the obvious direct definition has the simpler index Index o. Moreover, by using the more direct definition of the functorial map, the usual functor laws hold definitionally, as required. The proofs that return and join are natural transformations similarly hold by definition.

To describe the monadic structure of OpM O, we also require proofs of the coherence laws. We begin by giving an equivalent characterisation of the path space on OpM O X for any X : Type, which we use to construct the coherence witnesses. In particular, for every x y : OpM O X, the paths between x and y are trivially equivalent to the triples of a path p : Index x ≡ Index y on the indices, together with a heterogeneous path PathP (λ i → K (p i)) (Op x) (Op y) on the operations and a heterogeneous path PathP (λ i → $\llbracket$ $\mathcal{U}$ ∋ p i $\rrbracket$ → X) (Data x) (Data y) on the data. Moreover, for constructing the heterogeneous paths between the Data fields, we note that given any types X Y : Type, functions f : X → Y, g : Y → X and equivalence e : X ≃ Y, there is an equivalence between the heterogeneous path space PathP (λ i → ua e i → X) f g and families of paths (x : X) → f x ≡ g (fun e x).

▶ **Theorem 16** (OpM O is a monad on h-sets). *For any operad O : Operad $\mathcal{U}$ K, the assignment $X \mapsto$ OpM O X with unit return, multiplication join and functorial map <$> is a monad on the category of h-sets, with the monad laws witnessed by paths join-return$_1$, join-return$_2$ and join-assoc.*

Proof. We proceed by giving an overview of the path constructions witnessing the monadic laws; the details can be found in our formalisation. The required paths on indices are $\hat{\sum}$Idl, $\hat{\sum}$Idr and $\hat{\sum}$Assoc, and the corresponding heterogeneous paths on operations are then obtained from the operad laws idl O, idr O and assoc O. To construct the heterogeneous paths on data, we first substitute in the family $(\lambda\ p \to \mathsf{PathP}\ (\lambda\ i \to p\ i \to X))$ over the path interpretations $[\![\hat{\sum}\mathsf{Idl}]\!]$, $[\![\hat{\sum}\mathsf{Idr}]\!]$ and $[\![\hat{\sum}\mathsf{Assoc}]\!]$. From our earlier characterisation of heterogeneous paths of the form PathP $(\lambda\ i \to$ ua $e\ i \to X)\ f\ g$, it then suffices to construct families of paths $(x : X) \to f\ x \equiv g\ (\mathsf{fun}\ e\ x)$ for each monadic law. The identity laws hold definitionally, while associativity is proved in our formalisation. ◀

For any operad O : Operad $\mathcal{U}$ K, the algebras over the monad OpM O, or simply *the algebras over* O, describe how the abstract collection of operations represented by O can be interpreted as concrete operations on a carrier type.

▶ **Definition 17** (Operad algebra). *An* algebra *over an operad O : Operad $\mathcal{U}$ K is a pair of a carrier h-set A : Type together with an operad morphism $\alpha : O \Rightarrow$ Endo $\mathcal{U}$ A from O to the endomorphism operad on A (Definition 14).*

Examples of operad algebras include both the function $[\![\_]\!]$: IExpr $n \to$ (Fin $n \to$ Expr) $\to$ Expr with carrier Expr and the function fill : PartialList $A\ n \to$ (Fin $n \to$ List A) $\to$ List A with carrier List A. This notion of an operad algebra gives rise to an interpretation function runAlg : ($O \Rightarrow$ Endo $\mathcal{U}$ A) $\to$ OpM O $A \to A$ defined simply as runAlg α o = $\langle\ \alpha\ \rangle$ (Index o) (Op o) (Data o).

▶ **Example 18** (select and mapWhere via the partial-list operad). To conclude our discussion of monads over an operad, we provide a concrete example of using the monad over the partial list operad. We begin by letting $\hat{\mathbb{N}}$: Universe be the generalised operad universe of totally ordered finite sets, i.e. Code $\hat{\mathbb{N}} = \mathbb{N}$ and $[\![\ \hat{\mathbb{N}} \ni n\ ]\!] =$ Fin n, and PList A : Operad $\hat{\mathbb{N}}$ (PartialList A) be the partial list operad on an h-set A. We then observe that we can define monadic liftings of the _::_ and poke constructors. In particular, for every h-groupoid X : Type we construct the monadic lifting of _::_ as follows:

```
consM : A → OpM (PList A) X → OpM (PList A) X
consM a o = Index o ▷ (a :: Op o) ▷ Data o
```

We similarly define the monadic lifting of poke by:

```
pokeM : X → OpM (PList A) X → OpM (PList A) X
pokeM x o = suc (Index o) ▷ poke (Op o) ▷ λ { zero → x ; (suc i) → Data o i }
```

Using these two functions we can define a function that extracts elements from a list that satisfy a predicate, and preserve the original list with holes in place of these elements:

```
select : (A → Bool) → List A → OpM (PList A) A
select p [] = 0 ▷ [] ▷ λ ()
select p (a :: as) = if p a then pokeM a (select p as)  else consM a (select p as)
```

Intuitively, select highlights certain elements of a list, allowing us to modify them while leaving the rest intact. Indeed, if we let fill⇒ : PList A ⇒ Endo $\mathcal{U}$ (List A) be the operad morphism with ⟨ fill⇒ ⟩ = fill, then for every $p : A \to$ Bool and xs : List A we can construct a path of type runAlg fill⇒ ([_] <$> select p xs) ≡ xs, where [_] constructs a singleton list. That is, selecting elements and then putting them back unchanged gives the original list.

Building on select, we obtain a function that applies a given transformation to every element satisfying a predicate while leaving the rest of the list unchanged:

```
mapWhere : (A → Bool) → (A → A) → List A → List A
mapWhere p f xs = runAlg fill⇒ ((f ∘ [_]) <$> select p xs)
```

More generally, the monad over an operad gives a structured way to build effectful traversals that respect a chosen compositional structure on the underlying datatype: in this case, the partial-list operad structure on List A.

9 Free Operad

We have seen how a generalised form of operads that encompasses both planar and symmetric operads can be presented in HoTT. Using this framework, we now formalise one of the key operadic constructions, namely the *free* operad on a collection of operations. In particular, this is the left adjoint construction to the forgetful functor from operads to their underlying operations. Intuitively, the operations of free operads correspond to trees with nodes labelled by elements of a collection of operations depending on their arity, where composition is given by grafting of trees. Free operads allow us to construct operation trees independently from the choice of how to compose the underlying operations. In this section, we will give an account of the free construction for generalised operads in HoTT, together with some important properties and examples.

We begin by introducing the notion of a free planar operad on a countable family of h-sets $K : \mathbb{N} \to$ Type, and then show how the construction can be generalised. The operations of the free planar operad on K are characterised by planar, finitely-branching trees in which nodes with n-input vertices are labelled by an element of $K\ n$. We also require a unit tree with a single input and output vertex that acts as a left and right unit for tree composition.

For readers with experience of *rose trees*, the structure of the free planar operad may seem familiar. To recall, rose trees are planar finitely-branching trees, and are typically labelled with elements of a parameterised type. We can define rose trees as follows:

```
data RoseTree (A : Type) : Type where
  leaf : RoseTree A
  node : (n : ℕ) → A → (Fin n → RoseTree A) → RoseTree A
```

We can define a function count : RoseTree $A \to \mathbb{N}$ that counts leaves in a rose tree. The fibers of this function, i.e. the countable family of types mapping $n : \mathbb{N}$ to the type ∑[t ∈ RoseTree A] count $t \equiv n$ of rose trees with n leaves, has an operad structure given by tree composition with unit leaf. When A is the unit type, the induced operad is known as the *planar tree operad*. A generalisation of this construction involves labelling nodes with elements from a countable family of h-sets $K : \mathbb{N} \to$ Type, where a node with n-inputs is labelled with an element of $K\ n$: Type.

▶ **Definition 19** (Free planar operad on a countable family). *For any countable family* $K : \mathbb{N} \to$ *Type, the underlying operations of the* free planar operad *on* K *are given by the following inductive family:*

```
data FreePLOps (K : ℕ → Type) : ℕ → Type where
  unit : FreePLOps K 1
  comp : (ns : Fin n → ℕ) → K n → ((∀ i → FreePLOps K (ns i)) → FreePLOps K (Σ n ns)
```

This definition might at first appear rather different from that of rose trees. However, this is a consequence of indexing our inductive definition over the number of leaves rather than defining a separate count function whose fibers are equivalent to FreePLOps K.

As expected, our inductive construction for the operations of the free planar operad comes equipped with a planar operadic structure with the identity operation given by unit : FreePLOps K 1. While we do not give an account of the full operadic structure on FreePLOps here, it follows as a specific case of the more general free operad construction. In particular, given a universe $\mathcal{U}$: Universe, a first attempt at defining the operations of the free $\mathcal{U}$-operad is the following translation of FreePLOps to the generalised case:

```
data FreeOps (K : Code 𝒰 → Type) : Code 𝒰 → Type where
  leaf : FreeOps K (↑̂ 𝒰)
  node : (A : Code 𝒰) (B : ⟦ 𝒰 ∋ A ⟧ → Code 𝒰) →
         K A → (∀ a → FreeOps K (B a)) → FreeOps K (∑̂ 𝒰 A B)
```

However, this definition of FreeOps is not a sufficient characterisation of the free operad as it is not always a family of h-sets. This is not immediately evident as although node parametrises over a type of codes Code $\mathcal{U}$ that may not be an h-set, i.e. FinSet, these codes are then constrained by the output index of the node constructor. Indeed, it is possible to show that the assumption that FreeOps K A is an h-set for every family of h-sets K : Code $\mathcal{U} \to$ Type and every code A : Code $\mathcal{U}$ leads to a contradiction. In particular, for any universe $\mathcal{U}$ with an element $\hat{\bot}$: Code $\mathcal{U}$ such that the interpretation of $\hat{\bot}$ is the empty type, there is a retraction from FreeOps K $\hat{\bot}$ to Code $\mathcal{U}$. In turn, because retractions preserve h-levels, when Code $\mathcal{U}$ is not an h-set as in the case of FinSet, the assumption that FreeOps K $\hat{\bot}$ is an h-set yields a contradiction. More generally, for any $n \geq 0$, if Code $\mathcal{U}$ is an n-type then FreeOps K A is at most an n-type. The set-truncation path constructor we introduce below is therefore only required when Code $\mathcal{U}$ has higher path structure than an h-set.

As a consequence of FreeOps not being a family of h-sets, it is necessary to add a set-truncation path constructor

```
set : (A : Code 𝒰) → isSet (FreeOps K A)
```

to our inductive definition. Notably, this is not the same as defining a new family of types which maps each K : Code $\mathcal{U} \to$ Type and A : Code $\mathcal{U}$ to $\| \text{FreeOps } K\ A \|_0$, i.e. the set or 0-truncation of FreeOps K A. Moreover, in the case where the type of codes for the considered universe is an h-set, such as for the universe of totally-ordered finite sets, then the versions of the FreeOps with and without the set path constructor are equivalent.

By presenting the operations of the free operad on K as the higher-inductive family FreeOps K, the operad composition map requires explicit substitutions along paths on codes. In practice, this can lead to "transport hell", whereby operations built by composition are nested substitutions that cannot be unfolded in further computations. This is a notable issue when we are simply interested in computing along the tree structure. In the presence of (small) induction-recursion, this can be addressed by an alternative encoding that separates the shape of a tree from the indexing of its leaves.

Our alternative encoding uses the known equivalence between inductive families and small induction-recursion [17]. However, this equivalence has not been extended to higher-inductive families and higher small induction-recursion. To highlight this issue, we first consider a naive translation of only the data constructors of the higher inductive family FreeOps K to a small inductive-recursive definition.

▶ **Definition 20** (Inductive-recursive shape FreeOpsIR and its decoder CodeOp). *For every family* $K : \mathit{Code}\ \mathcal{U} \to \mathit{Type}$*, the type* $\mathit{FreeOpsIR}\ K : \mathit{Type}$ *and the recursive function* $\mathit{CodeOp}\ K : \mathit{FreeOpsIR}\ K \to \mathit{Code}\ \mathcal{U}$ *are defined mutually as follows:*

$$\begin{aligned}
&\mathit{data}\ \mathit{FreeOpsIR}\ K : \mathit{Type}\ \mathit{where}\\
&\quad \mathit{leaf} : \mathit{FreeOpsIR}\ K\\
&\quad \mathit{node} : (A : \mathit{Code}\ \mathcal{U}) \to K\ A \to (⟦\ \mathcal{U} \ni A\ ⟧ \to \mathit{FreeOpsIR}\ K) \to \mathit{FreeOpsIR}\ K
\end{aligned}$$

$$\begin{aligned}
&\mathit{CodeOp}\ K\ \mathit{leaf} = \hat{\top}\ \mathcal{U}\\
&\mathit{CodeOp}\ K\ (\mathit{node}\ A\ k\ ts) = \hat{\sum}\ \mathcal{U}\ A\ (\mathit{CodeOp} \circ ts)
\end{aligned}$$

In particular, there is an equivalence between fibers of CodeOp K and the inductive family FreeOps K without its set path constructor. However, as the type of codes Code $\mathcal{U}$ may not be an h-set, the fibers of CodeOp K cannot always be shown to be h-sets and are hence not equivalent to the higher-inductive family FreeOps K.

In the absence of the axiom of choice, it is not sufficient to simply consider the set truncated fibers over CodeOp K, i.e. the family mapping each code A : Code $\mathcal{U}$ to the 0-truncated type $\| \sum[\ t \in \mathsf{FreeOpsIR}\ K\ ]\ \mathsf{CodeOp}\ K\ t \equiv A \|_0$. Instead, we require the addition of an appropriate higher path constructor to FreeOpsIR K, together with the action of CodeOp K on this constructor, to ensure that the fibers of CodeOp K are h-sets. To do this, we begin by noting that for any types X, Y : Type and function $f : X \to Y$, the proposition that the fibers of f are all h-sets is equivalent to the proposition that the functorial action of f on paths is an embedding, i.e. for all terms $x_1\ x_2 : X$ and paths $p\ q : x_1 \equiv x_2$, the function $\mathsf{cong}\ (\mathsf{cong}\ f) : p \equiv q \to \mathsf{cong}\ f\ p \equiv \mathsf{cong}\ f\ q$ is an equivalence. Indeed, this embedding is precisely what we will aim to establish by adding a path constructor to FreeOpsIR K.

To construct this embedding, it suffices to show that $\mathsf{cong}\ (\mathsf{cong}\ (\mathsf{CodeOp}\ K)) : p \equiv q \to \mathsf{cong}\ (\mathsf{CodeOp}\ K)\ p \equiv \mathsf{cong}\ (\mathsf{CodeOp}\ K)\ q$ is an isomorphism. We can introduce an equivalent isomorphism into FreeOpsIR using a path constructor whose type is almost identical to that of $\mathsf{cong}\ (\mathsf{cong}\ (\mathsf{CodeOp}\ K))$. However, to ensure FreeOpsIR only appears strictly positively, we first flip the resulting square along its diagonal. Concretely, we introduce a path constructor that for each $t\ u$: FreeOpsIR and $p\ q : t \equiv u$, constructs the following path:

$$\mathsf{set}\ p\ q : \mathsf{PathP}\ (\lambda\ i \to \mathsf{CodeOp}\ K\ (p\ i) \equiv \mathsf{CodeOp}\ K\ (q\ i))\ \mathsf{refl}\ \mathsf{refl} \to p \equiv q$$

Interestingly, it is possible to show that the set $p\ q$ path constructor is both a section and retraction of $\mathsf{cong}\ (\mathsf{cong}\ (\mathsf{CodeOp}\ K))$, which yields the following.

▶ **Proposition 21** (Fibers of CodeOp K are h-sets). *For every universe* $\mathcal{U}$ *and family* K : $\mathit{Code}\ \mathcal{U} \to \mathit{Type}$*, the functorial action of* $\mathit{CodeOp}\ K$ *on paths is an embedding. Equivalently, for every code* $A : \mathit{Code}\ \mathcal{U}$ *the fiber* $\mathit{fiber}\ (\mathit{CodeOp}\ K)\ A$ *is an h-set.*

Proof. This is a generalisation of *univalent inductive-recursive* types. A univalent inductive-recursive definition is one in which the mutually defined recursive function is forced to have propositional fibers by adding a path constructor that makes its functorial action on paths an embedding; the resulting type behaves like a Tarski universe whose decoding map reflects

path structure. Our setting generalises this from propositional fibers to set-level fibers, which is what is required when the codomain Code $\mathcal{U}$ is itself a groupoid rather than a set. The details of this proof can be found in our Cubical Agda formalisation. ◀

Notably, we can always eliminate terms of FreeOpsIR K into any h-set whereby the path constructor set is necessarily respected. Importantly, this includes eliminating into the fibers of the function CodeOp K which is necessary to define the composition map for the free operad. To construct this, we first introduce the following two required families of paths:

$$\hat{\sum}\text{IdlD}\ \mathcal{U}\ X : X\ (\text{inv}\ (\llbracket \hat{\top} \rrbracket\ \mathcal{U})\ \text{tt}) \equiv \hat{\sum}\ \mathcal{U}\ (\hat{\top}\ \mathcal{U})\ X$$

$$\hat{\sum}\text{AssocD}\ \mathcal{U}\ A\ B\ C :$$

$$\hat{\sum}\ \mathcal{U}\ A\ (\lambda\ a \to \hat{\sum}\ \mathcal{U}\ (B\ a)\ \lambda\ b \to C\ (\text{inv}\ (\llbracket \hat{\sum} \rrbracket\ \mathcal{U}\ A\ B)\ (a\ ,\ b))) \equiv \hat{\sum}\ \mathcal{U}\ (\hat{\sum}\ \mathcal{U}\ A\ B)\ C$$

where $X : \llbracket\ \mathcal{U} \ni \hat{\top}\ \rrbracket \to \text{Code}\ \mathcal{U}$, $A : \text{Code}\ \mathcal{U}$, $B : \llbracket\ \mathcal{U} \ni A\ \rrbracket \to \text{Code}\ \mathcal{U}$ and $C : \llbracket\ \mathcal{U} \ni \hat{\sum}\ \mathcal{U}\ A\ B\ \rrbracket \to \text{Code}\ \mathcal{U}$. We do not give the full constructions for these paths here, but they follow from $\hat{\sum}$Idl and $\hat{\sum}$Assoc. We proceed by defining a variant of the operad composition map whose input operations are not indexed. For each tree t : FreeOpsIR K and family of trees $ts : \llbracket\ \mathcal{U} \ni \text{CodeOp}\ K\ t\ \rrbracket \to \text{FreeOpsIR}\ K$ we construct the following term:

```
graft t ts : fiber (CodeOp K) (∑̂ U (CodeOp K) (CodeOp K ∘ ts))
graft leaf ts = ts (inv ([[⊤̂]] U) tt) , ∑̂IdlD U (CodeOp K ∘ ts) (inv ([[⊤̂]] U) tt)
graft (node A k ts) tss
  = let iv = inv ([[∑̂]] U A (CodeOp K ∘ ts))
        us = graft (ts a) λ b → tss (iv (a , b))
        ac = ∑̂AssocD U (CodeOp K ∘ ts) (CodeOp K ∘ tss)
    in node A k (fst ∘ us) , (λ i → ∑̂ U A λ a → snd (us a) i) · ac)
```

We note that the action of graft on the set path constructor follows from the proof that fibers of CodeOp K are h-sets. The composition map for the free operad follows from the definition of graft by reinserting the indexing of the input operations. The unit operation is given by the term leaf whereby CodeOp K leaf $\equiv \hat{\top}\ \mathcal{U}$ holds definitionally. Finally, while we do not provide the constructions for the paths witnessing the identity and associativity operad laws here, they can be found in our Cubical Agda formalisation [4] and involve significant manipulation of nested substitutions.

▶ **Definition 22** (Operad on fiber (CodeOp K))**.** *Given a $\mathcal{U}$-species K : Code $\mathcal{U} \to$ Type, FreeOperad K : Operad $\mathcal{U}$ (fiber (CodeOp K)) is the operad whose composition map comp $A\ B$ is defined by mapping every (t, p) : fiber (CodeOp K) A to the transport of graft along p in the type $((a : \llbracket\ \mathcal{U} \ni A\ \rrbracket) \to$ fiber (CodeOp K) $(B\ a)) \to$ fiber (CodeOp K) $(\hat{\sum}\ \mathcal{U}\ A\ B)$.*

We have thus far described the operadic structure on the fibers of CodeOp K for every $\mathcal{U}$-species K : Code $\mathcal{U} \to$ Type, but have not yet shown it is the *free* operad on K. To do so, we first highlight the categories on which we will define the free adjunction. In particular, the objects of our underlying category are $\mathcal{U}$-species which can be understood as functors from the h-groupoid structure of Code $\mathcal{U}$ to the category of h-sets and functions. Concretely, the functorial action of a $\mathcal{U}$-species K : Code $\mathcal{U} \to$ Type is given by path substitution in K, i.e. a path $p : A \equiv B$ is lifted to subst $K\ p : K\ A \to K\ B$. It is a standard result of path substitution that this satisfies the functorial laws. For any two $\mathcal{U}$-species $K\ L$: Code $\mathcal{U} \to$ Type, their hom is given by the type of natural transformations $(A : \text{Code}) \to K\ A \to L\ A$, where naturality follows from substitution commuting with slice morphisms. Finally, the target category for the free construction of a $\mathcal{U}$-operad is precisely the category of $\mathcal{U}$-operads from Section 7.

There is a forgetful functor from the category of $\mathcal{U}$-operads to the category of $\mathcal{U}$-species which simply forgets the operadic structure and acts similarly on operad morphisms by means of the projection $\langle \_ \rangle : (O^K \Rightarrow O^L) \to (A : \mathsf{Code}) \to K\ A \to L\ A$. The free operad on a $\mathcal{U}$-species $K : \mathsf{Code}\ \mathcal{U} \to \mathsf{Type}$ is, up to a path, the operad constructed by the left-adjoint to this forgetful functor. As we will show, the left-adjoint is the functor mapping each $K : \mathsf{Code}\ \mathcal{U} \to \mathsf{Type}$ to the collection of operations $\mathsf{fiber}\ (\mathsf{CodeOp}\ K) : \mathsf{Code}\ \mathcal{U} \to \mathsf{Type}$ equipped with its previously described operadic structure. The functoriality of this construction follows from the universal property of being a left adjoint in the usual way. The first step in establishing that adjunction is to define the following unit natural transformation:

$$\eta : (A : \mathsf{Code}\ \mathcal{U}) \to K\ A \to \mathsf{fiber}\ (\mathsf{CodeOp}\ K)\ A$$
$$\eta\ A\ k = \mathsf{node}\ A\ k\ (\lambda\ a \to \mathsf{leaf})\ ,\ \hat{\sum}\mathsf{Idr}\ \mathcal{U}\ A$$

Moreover, we require that for every operad $L : \mathsf{Code}\ \mathcal{U} \to \mathsf{Type}$, $O : \mathsf{Operad}\ \mathcal{U}\ L$, and every morphism of $\mathcal{U}$-species $f : (A : \mathsf{Code}\ \mathcal{U}) \to K\ A \to L\ A$, that we can construct an operad morphism from the operadic structure $\mathsf{CodeOp}\ K$ to O. In particular, we begin by defining a family of morphisms $\mathsf{interpret}\ O\ f : (A : \mathsf{Code}\ \mathcal{U}) \to \mathsf{fiber}\ (\mathsf{CodeOp}\ K)\ A \to L\ A$ as follows:

$$\mathsf{interpret}\ O\ f\ .(\hat{\uparrow}\ \mathcal{U})\ \mathsf{leaf} = \mathsf{id}\ O$$
$$\mathsf{interpret}\ O\ f\ .(\hat{\sum}\ \mathcal{U}\ A\ B)\ (\mathsf{node}\ A\ B\ k\ ts) = \mathsf{comp}\ O\ A\ B\ (f\ A\ k)\ \lambda\ a\ \to \mathsf{interpret}\ O\ f\ (B\ a)\ (ts\ a)$$

Note that $\mathsf{interpret}$ definitionally respects the operad identity. The proof that it also respects operad composition and thus extends to a morphism between the operads $\mathsf{FreeOperad}\ K$ and O requires manipulation of nested substitutions and is provided in our library.

▶ **Theorem 23** (Free–forgetful adjunction). *For every operad $O : \mathsf{Operad}\ \mathcal{U}\ L$ and every morphism of $\mathcal{U}$-species $f : (A : \mathsf{Code}\ \mathcal{U}) \to K\ A \to L\ A$, the type*

$$\sum[\ g \in \mathsf{FreeOperad}\ K\ ]\ ((A : \mathsf{Code}\ \mathcal{U})\ (k : K\ A) \to \langle\ g\ \rangle\ A\ (\eta\ A\ k) \equiv f\ A\ k)$$

is contractible, with the base point given by the function $\mathsf{interpret}\ O\ f$ extended to an operad map together with the proof that $\mathsf{interpret}\ O\ f\ A\ (\eta\ A\ k) = f\ A\ k$ for all $A : \mathsf{Code}\ \mathcal{U}$ and $k : K\ A$. Consequently, the $\mathsf{FreeOperad}$ construction is left adjoint to the forgetful functor U from $\mathcal{U}$-operads to $\mathcal{U}$-species.

Proof. The detailed construction is provided in our library. The category of $\mathcal{U}$-species can be understood as a category of *signatures* over the universe of $\mathcal{U}$-arities: an object specifies, for each code $A : \mathsf{Code}\ \mathcal{U}$, the type of generating operations of that arity. The adjunction therefore exhibits $\mathsf{FreeOperad}$ as the canonical way to turn a signature of generators into an operad whose laws are freely generated. ◀

10 Related Work

Several generic representations of datatypes have been developed in the type theory literature. The idea of separating structure from data first arose in work on shapely types [18]. More recently, containers were developed to capture strictly positive types [1]. The more general presentation of indexed containers was later developed to capture inductive families [5]. Containers are closed under finite products and exponentials [6], and have a notion of derivative [3]. The theory of containers also captures the shapely types. In particular, shapely type constructors are precisely given by the extensions of discretely-finite containers.

A key aspect of our theory of internal operads is the notion of a finite set, and the choice of a suitable definition of finiteness. Such a choice only manifests in constructive mathematics, where various classically equivalent notions of finiteness are distinct. In this

paper we adopt the notion of finiteness used in previous work on internalising the theory of combinatorial species in HoTT [26], namely Bishop-finiteness. However, there has also been work on internalising "enumerated" types and Kuratowski finite sets in the framework of HoTT [13, 15]. Adopting a different notion of finiteness impacts both which collection of operations are definable as a species, and which proofs about operads can be internalised.

Combinatorial species were introduced to unify approaches for analysing generating functions of discrete structures [20]. Intuitively, the idea of a species arose from the idea of building a structure from a finite collection of labels. Our internalisation of operads in HoTT can be seen as an extension of previous work on internalising combinatorial species [26]. In particular, Yorgey internalises combinatorial species to capture specific classes of data types, analogous to shapely types and the theory of containers.

Coloured operads generalise ordinary operads by labelling the inputs and outputs of operations with elements from a set C, and present a well-behaved notion of composition for finitary C-typed operations [25]. Such coloured operads have been formalised and studied in Coq [14]. In this paper, we focus on the single-coloured case and pursue a different direction of generalisation: rather than varying the set of colours, we vary the groupoid indexing the operations of an operad. This approach provides a uniform way to capture different flavours of operads by encoding symmetries directly in the indexing groupoid.

The theory of operads originated in the field of algebraic topology [21, 8] for describing operations on iterated loop-spaces. Another direction generalising operads is the theory of opetopes and opetopic sets [7], which have also been considered in the context of type theory by [22]. The Baez-Dolan slicing construction describes how to construct n-opetopes as an iterated construction making use of the free coloured operad. Generalising our work to the coloured setting would allow us to internalise this construction as an alternative way to define opetopes in HoTT.

11 Conclusion and Further Work

In this paper we showed how operads can be internalised in HoTT, and how this gives rise to a generic calculus of operations. Notably, we showed how discretely finite containers, which represent inductive data types with constructors of finite arity, have a natural operadic interpretation. This offers an operation-centric view of inductive types, complementing the traditional data-centric perspective. Our results open up a new line of work on investigating properties and applications of operads from a type-theoretic viewpoint. From a practical perspective, operad algebras provide semantics for embedded domain-specific languages whose terms are finitary compositions over a chosen signature: the free operad gives the syntax, and operad-algebra morphisms give compositional interpreters. Interesting topics for further work include generalising from species and operads to the "coloured" counterparts, considering how the notion of finiteness that we adopt influences which algebraic structures are expressible as a family over finite sets, and dualising our ideas to co-operads.

References

1 Michael Abbott. *Categories of Containers*. PhD thesis, University of Leicester, 2003.

2 Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers: Constructing Strictly Positive Types. *Theoretical Computer Science*, 342(1), 2005. doi:10.1016/j.tcs.2005.06.002.

3 Michael Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. Derivatives of Containers. In *International Conference on Typed Lambda Calculi and Applications*. Springer, 2003.

4 HoTT Operads Cubical Agda Library. https://tinyurl.com/y5fbrzdb, 2026.
5 Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor McBride, and Peter Morris. Indexed Containers. *Journal of Functional Programming*, 25, 2015. doi:10.1017/S095679681500009X.
6 Thorsten Altenkirch, Paul Levy, and Sam Staton. Higher-Order Containers. In *Conference on Computability in Europe*. Springer, 2010.
7 John C Baez and James Dolan. Higher-Dimensional Algebra III: n-Categories and the Algebra of Opetopes. *Advances in Mathematics*, 135(2):145–206, 1998.
8 John Michael Boardman and Rainer M Vogt. *Homotopy Invariant Algebraic Structures on Topological Spaces*, volume 347. Springer, 2006.
9 Jacques Carette, Chao-Hong Chen, Vikraman Choudhury, and Amr Sabry. From Reversible Programs to Univalent Universes and Back. *Electronic Notes in Theoretical Computer Science*, 336:5–25, 2018. doi:10.1016/j.entcs.2018.03.013.
10 Vikraman Choudhury, Jacek Karwowski, and Amr Sabry. Symmetries in Reversible Programming: From Symmetric Rig Groupoids to Reversible Programming Languages. *Proceedings of the ACM on Programming Languages*, 6(POPL), 2022. doi:10.1145/3498667.
11 J. Daniel Christensen. A Characterization of Univalent Fibrations, 2022.
12 Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom. In Tarmo Uustalu, editor, *21st International Conference on Types for Proofs and Programs (TYPES 2015)*, volume 69 of *Leibniz International Proceedings in Informatics (LIPIcs)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.TYPES.2015.5.
13 Thierry Coquand and Arnaud Spiwack. Constructively Finite? In *Contribuciones Científicas en Honor de Mirian Andrés Gómez*. Universidad de La Rioja, 2010.
14 Zachary Flores, Angelo Taranto, Eric Bond, and Yakir Forman. A Formalization of Operads in Coq, 2023. doi:10.48550/arXiv.2303.08894.
15 Dan Frumin, Herman Geuvers, Léon Gondelman, and Niels van der Weide. Finite Sets in Homotopy Type Theory. In *International Conference on Certified Programs and Proofs*, 2018.
16 Håkon Robbestad Gylterud. Symmetric Containers. Master's thesis, University of Oslo, 2011., 2011.
17 Peter Hancock, Conor McBride, Neil Ghani, Lorenzo Malatesta, and Thorsten Altenkirch. Small Induction Recursion. In *International Conference on Typed Lambda Calculi and Applications*, pages 156–172. Springer, 2013. doi:10.1007/978-3-642-38946-7_13.
18 C. Barry Jay and J. Robin B. Cockett. Shapely Types and Shape Polymorphism. In *European Symposium on Programming*. Springer, 1994.
19 Philipp Joram and Niccolò Veltri. Data Types with Symmetries via Action Containers. In *30th International Conference on Types for Proofs and Programs (TYPES 2024)*, pages 6–1. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPIcs.TYPES.2024.6.
20 André Joyal. Une Théorie Combinatoire des Séries Formelles. *Advances in Mathematics*, 42(1), 1981.
21 J Peter May. *The Geometry of Iterated Loop Spaces*, volume 271. Springer, 2006.
22 Pierre-Louis Curien, Cédric Ho Thanh, and Samuel Mimram. Syntactic Approaches to Opetopes, 2019. arXiv:1903.05848.
23 Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types. *Proceedings of the ACM on Programming Languages*, 3(ICFP), 2019. doi:10.1145/3341691.
24 Vladimir Voevodsky, Benedikt Ahrens, Daniel Grayson, et al. UniMath: A Computer-Checked Library of Univalent Mathematics. Available at https://github.com/UniMath/UniMath.
25 Donald Yau. *Colored Operads*, volume 170. American Mathematical Society, 2016.
26 Brent Abraham Yorgey. *Combinatorial Species and Labelled Structures*. PhD thesis, University of Pennsylvania, 2014.