HoTT Operads

Brandon Hewer
University of Nottingham

Graham Hutton
University of Nottingham

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 operads 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.

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 [14]. 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 to 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. More concretely, we:

- Internalise the notion of both planar (section 4) and symmetric (section 5) operads in homotopy type theory, and provide practical examples of each form;
- Prove that every operad induces a canonical monad (section 9), which provides an *operadic* style of constructing programs from a small collection of composable terms;
- Demonstrate how the free operad can be constructed as a higher inductive family (section 10).

The paper is aimed at programming language theorists who are interested in programming generically over classes of operations, and type/category theorists who wish to reason about operads in a proof assistant. All our results are formalised in Cubical Agda [6].

2 BACKGROUND AND METATHEORY

Throughout this paper, ideas and formalisations will be presented in the metatheory of Cubical Agda. We begin by reviewing the idea of *path types*, a key concept in homotopy type theory (HoTT). In particular, for any type A, we can construct the type $x \equiv y$ of *paths* between two 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 [8], 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 $M : (y : A) \to x \equiv y \to \text{Type}$, if we have a proof $t : M\ x\ \textbf{refl}$ 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 reflection on x.

Cubical Agda also provides a primitive construction for heterogenous path types which are presented in the form PathP $(\lambda\ i \to T)\ a\ b$ for terms $a : A$, $b : B$ and T : Type. The first argument to PathP is a continuous function out of the built-in interval type I for which the endpoints i0 i1 : 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 : Type → 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 understood as operations on Expr, where val is an algebra on the constant functor choosing ℕ, and add is a (curried) algebra on 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 more specifically compose. For example, consider the collection of operations constructed as finite compositions of the constructors val and add, together with the identity function id : Expr → Expr. Importantly, the inclusion of the identity function allows us to introduce variables in operations, e.g. in the term $\lambda\ x\ y\ z \to$ add (add x (val 1)) (add y z). Operations constructed in this way are always finite and linear.

To represent the desired collection of operations on Expr, we first observe that the type ∑[$n \in \mathbb{N}$] ((Fin n → Expr) → Expr), where Fin : 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 : 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 : 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, we first require a family of functions sum n : (Fin n → ℕ) → ℕ for calculating the sum of n natural numbers, which can be given inductively as follows:

```
sum 0 ns = 0
sum 1 ns = ns zero
sum (suc (suc n)) ns = ns zero + sum (suc n) (λ i → ns (suc i))
```

For every n : ℕ and n-ary product of natural numbers ns : Fin n → 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 : sum m (π^1 ns) + sum n (π^2 ns) ≡ 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 given any n-ary operation, either plugging its output into the input of the identity operation represented, or plugging the identity's output into each of its n inputs, leaves the operation unchanged. Associativity asserts 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 (Fin n → Expr) → Expr, where the composition map is simply given by n-ary function composition, and the unit operation is given by the term λ es → es zero : (Fin 1 → Expr) → Expr. The interpretation function ⟦_⟧ : IExpr n → (Fin n → Expr) → 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 the 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 several important properties of our definition.

We begin by recalling that a countable family of types K : ℕ → Type can be understood as a family of finitary operations indexed by their arities. To avoid higher coherences, we consider only families of h-sets, i.e. K : ℕ → 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 ∑[$A \in$ Type] isSet A, where isSet A is the proposition that A is an h-set. However, given a family of sets K : ℕ → hSet, we will write K n : Type for the first projection, and make it clear when we use the proof that this type is an h-set.

Given any family of h-sets $K : \mathbb{N} \to \mathsf{hSet}$, we can define the planar operads on K as a record type record $\mathsf{NonSymmOperad}\ K$: Type, i.e. a finitely iterated sigma type with named projections. In particular, a planar operad $O : \mathsf{NonSymmOperad}\ K$ comes equipped with 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 $ns : \mathsf{Fin}\ n \to \mathbb{N}$. The composition map of an operad 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

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

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

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

where $\mathsf{sum\text{-}idr}\ n : \mathsf{sum}\ n\ \ (\lambda\ i \to 1) \equiv n$ witnesses that the sum of n ones is n. Intuitively, $\mathsf{idr}\ O\ n\ k$ 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,

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

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

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

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

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

and bottom-up composition

$$\mathsf{comp}\ O\ n\ (\mathsf{sum}\ n\ ns)\ (\mathsf{uncurry}\ nss \circ \mathsf{sum}\Sigma\ ns)$$
$$(\mathsf{comp}\ O\ n\ ns\ k\ ks)\ (\mathsf{uncurry}\ kss \circ \mathsf{sum}\Sigma\ ns)$$

We have already seen one example of a planar operad, namely the family of h-sets $\mathsf{IExpr} : \mathbb{N} \to \mathsf{hSet}$ equipped with the n-ary composition map comp described in the previous section. As a second example, we will consider *partial lists* with ‘holes’:

```
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 $\mathsf{PartialList}\ A$ 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 $\_{+}{+}\_ : \mathsf{PartialList}\ A\ m \to \mathsf{PartialList}\ A\ n \to \mathsf{PartialList}\ A\ \ (m + n)$, defined by:

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

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

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

with n-ary composition map defined as follows:

```
comp O 0 ns xs xss = xs
comp O (suc n) ns (x :: xs) xss = x :: comp O (suc n) ns xs xss
comp O 1 ns (poke xs) xss =
  subst (PartialList A) (+-zero (ns zero)) (xss zero ++ xs)
comp O (suc (suc n)) ns (poke xs) xss =
  xss zero ++ comp O (suc n) (ns ∘ suc) xs (xss ∘ suc)
```

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 consider whether the operations characterised by the operadic structure on $\mathsf{PartialList}\ A$ 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. Similarly to the map $[\![\_]\!] : \mathsf{IExpr}\ n \to (\mathsf{Fin}\ n \to \mathsf{Expr}) \to \mathsf{Expr}$ in the previous section, fill can be shown to be a homomorphism between planar operads and respects the compositional structure and identity operation. We discuss how operads homomorphisms can be internalised in HoTT in Section 8.

5 SYMMETRIC OPERADS

In the previous section we introduced planar operads in HoTT, as a generalisation of the usual notion of operads whose operations also come equipped with actions of symmetric groups that are compatible with the operad 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 $O : \mathsf{NonSymmOperad}\ K$ 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 isomorphisms between the h-sets A and B. An isomorphism $\sigma : A \simeq B$ 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, $\mathsf{permute}\ O\ n$ 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 an additional field witnessing that $\mathsf{permute}\ O\ n$ is compatible with the operad composition map:

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

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 a singleton and consequently 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 in HoTT, known as Bishop-finite sets, internalises finiteness as a predicate on types, $\mathsf{isFinite} : \mathsf{Type} \to \mathsf{Type}$, as follows:

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

where $\|\ X\ \|$ is the propositional truncation of a type X. 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.

The universe of Bishop-finite sets can be internalised à la Tarski as a large type of codes $\mathsf{FinSet} = \sum[\ A \in \mathsf{Type}\ ]\ \mathsf{isFinite}\ A$, together with an interpretation map $\mathsf{El} : \mathsf{FinSet} \to \mathsf{Type}$ that is given by the first projection map. 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. 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 finite set $(A\ ,\ n\ ,\ p) : \mathsf{FinSet}$, it follows that from the truncated isomorphism 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 isomorphisms $\mathsf{El}\ A \simeq \mathsf{El}\ B$. We will express the forward direction of this equivalence as follows:

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

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 Voevodsky's unimath Coq library. 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 $A \hat{\equiv} B : \mathsf{FinSet}$ with a definitional equality $\mathsf{El}\ (A \hat{\equiv} B) = \mathsf{El}\ A \simeq \mathsf{El}\ B$ 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-set. Indeed, such families are also known as *combinatorial species*, or equivalently, *finitary containers*. 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:

$$\begin{aligned} \hat{\textstyle\sum}\text{-idr}\ A &: \hat{\textstyle\sum}\ A\ \ (\lambda\ a \to \top)\ \equiv A \\ \hat{\textstyle\sum}\text{-assoc}\ A\ B\ C &: \hat{\textstyle\sum}\ A\ (\lambda\ a \to \hat{\textstyle\sum}\ \ (B\ a)\ \ (C a)) \\ &\equiv \hat{\textstyle\sum}\ (\hat{\textstyle\sum}\ A\ B)\ (\lambda\ (a\ ,\ b) \to C\ a\ b) \end{aligned}$$

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 $\hat{\sum}\text{-idr}$ and $\hat{\sum}\text{-assoc}$ by applying un to the following canonical isomorphisms:

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

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

```
idl O A k : PathP (λ i → K (Σ̂-idl A i))
                  (comp O A (λ a → ⊤) k (λ a → id O)) k,
```

where the path $\hat{\Sigma}$-idl $A : \hat{\Sigma}\ \top\ (\lambda\ t \to A) \equiv A$ is constructed by the application of un $(\hat{\Sigma}\ \top\ (\lambda\ t \to A))\ A$ to the canonical isomorphism that exists between $\top \times$ El A and 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 : FinSet, family of finite sets B : El $A \to$ FinSet, operation $k : K\ A$, family of operations $ks : (a :$ El $A) \to K$ (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. 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 $K\ L$: FinSet $\to$ Type, family of maps $(A :$ FinSet$) \to K\ A \to L\ A$, operation $k : K\ A$, and symmetric group element $\sigma : A \equiv A$, a proof of $f\ A$ (subst $K\ \sigma\ k$) $\equiv$ subst $L\ \sigma\ (f\ A\ k)$ follows directly from substitution commuting with morphisms in slice categories.

We now recall our example of the planar operadic structure on the type family IExpr : $\mathbb{N} \to$ Type, and introduce the following, near identical, example of a FinSet indexed family:

```
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, El $(A \uplus B) =$ El $A \uplus$ El B, and $\bot$ is the empty type, i.e. the finite set with cardinality 0. The proof that finite sets are closed under finite coproducts can again be found both in our Cubical Agda formalisation and in Voevodsky's unimath library. Notably, SymExpr is a family of large types, which is a consequence of indexing on the large type FinSet and the constructor add↑ being parameterised over a large type. To address this, in Section 6 we describe how to define a small type of unordered finite sets.

In order 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:

```
⊎-distr A B C : (Σ[ a ∈ A ] C (inl a)) ⊎ (Σ[ b ∈ B ] C (inr b))
                ≃ (Σ[ x ∈ A ⊎ B ] C x)
```

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

```
Σ-idl A : A tt ≃ Σ[ x ∈ ⊤ ] A x
  Σ-⊥ B : ⊥ ≃ Σ[ x ∈ ⊥ ] 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)) (Σ[ x ∈ ⊤ ] B x) (Σ-idl B)) (es tt)
comp O .⊥ B (val↑ n) es =
  subst SymExpr (un ⊥ (Σ[ x ∈ ⊥ ] B x) (Σ-⊥ 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 SMALL FINSET

In contrast to the typical inductive presentation of $\mathbb{N}$, we have presented the universe of finite sets FinSet as a *large* type. Consequently, many of the FinSet-indexed families we construct, such as SymExpr, will themselves necessarily be families of large types. Moreover, it is also necessary to address this size issue in the definition of a symmetric operad, that will instead need to be defined over large families of types of the form FinSet $\to$ Type$_1$. However, as we will demonstrate, this size issue arises as a consequence of the choice of representation of FinSet. In this section, we introduce two small equivalent encodings of FinSet, the first as a higher-inductive type and the second using higher-induction recursion.

Intuitively, the underlying type of points or *connected components* of FinSet is equivalent to the natural numbers, i.e. we can construct an equivalence between the set-truncation of FinSet and the type $\mathbb{N}$. Another way to understand this connection is by constructing the universe of finite sets by starting with $\mathbb{N}$ and 'adding' in paths corresponding to the permutations of finite sets. This describes a *groupoid quotient* of the natural numbers by the transitive relation mapping $m\ n : \mathbb{N}$ to the h-set of permutations Fin $m \simeq$ Fin n. A groupoid quotient is a known higher-inductive construction in homotopy type theory, and we omit the details here.

Unfortunately, in practice it can be challenging and inefficient to define eliminators on this definition of finite sets, due to the added path constructors. In a type theory with higher-induction recursion, we can instead encode finite sets by mutually defining a higher-inductive type FinSet : Type together with an interpretation function El : FinSet → Type, where FinSet is given as follows:

```
data FinSet : Type where
  size : ℕ → FinSet
  un : (A B : FinSet) → El A ≃ El B → A ≡ B
```

The function El is then mutually defined by

```
El (size n) = Fin n
El (un A B p i) = ua (El A) (El B) p i
```

where for any types X and Y, the function ua X Y : $X \simeq Y \to X \equiv Y$ is the inverse of the canonical map from paths to equivalences pathToEquiv X Y : $X \equiv Y \to X \simeq Y$, as follows from univalence.

This small higher-inductive recursive representation of FinSet is equivalent to our original encoding as a large type. The construction relies on the fact that for any finite sets A B : FinSet, the function un A B : El $A \simeq$ El $B \to A \equiv B$ is inverse to the composition of the functorial action of El on paths with the canonical map from paths to equivalences, i.e. the function pathToEquiv (El A) (El B) ∘ cong El. Consequently, it can be shown that El is in an embedding, i.e. the functorial action of El on paths establishes an equivalence between the path spaces El $A \equiv$ El B and $A \equiv B$. Equivalently, the function El has propositional fibers, which means that for any X : Type the type ∑[A ∈ FinSet] El $A \equiv X$ is an h-proposition. The proof that El has propositional fibers can then be used to construct a family of isomorphisms between the fibers of El and the propositional family isFinite : Type → Type. By contractibility of singletons, the dependent sum taken over the fibers of El is equivalent to FinSet, and therefore we can establish the desired equivalence.

In this paper, we proceed by using the higher-inductive recursive definition of FinSet of finite sets, which is considerably easier to work with in practice. However, when working in a type theory with higher-inductive types but without higher-induction recursion, the issue of size can instead be addressed by using the groupoid quotient encoding discussed in this section.

7 GENERALISED OPERAD UNIVERSES

We have seen how symmetric and planar operads can be defined in HoTT by varying the Tarksi universe that indexes operations. For the planar case, we considered operations indexed by the universe of totally-ordered finite sets (ℕ, Fin), while for the symmetric case we considered the more general groupoid of finite sets and bijections (FinSet, 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 ⊤. 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 Tarksi 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 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

$$\hat{\sum} : (A : \text{Code}\ \mathcal{U}) \to (⟦\ \mathcal{U} \ni A\ ⟧ \to \text{Code}\ \mathcal{U}) \to \text{Code}\ \mathcal{U}$$

such that for every A : Code $\mathcal{U}$ and family B : ⟦ $\mathcal{U} \ni A$ ⟧ → Code $\mathcal{U}$, we have the following equivalance:

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

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

$$\hat{\sum}\text{-assoc}\ A\ B\ C : \hat{\sum}\ A\ (\lambda\ a \to \hat{\sum}\ (B\ a)\ (C\ a))$$
$$\equiv \hat{\sum}\ (\hat{\sum}\ A\ B)\ (\text{uncurry}\ C \circ ⟦\hat{\sum}⟧\ A\ B).$$

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 : Code $\mathcal{U}$, we require the following extra field:

$$\text{Inj}\ A\ B : ⟦\ \mathcal{U} \ni A\ ⟧ \simeq ⟦\ \mathcal{U} \ni B\ ⟧ \to A \equiv B$$

However, the Inj field alone is insufficient to map the higher path structure between underlying types in a universe to the the corresponding higher path structure between their codes In particular, we recall 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 $\mathcal{U}$: Universe, a $\mathcal{U}$-operad has operations given by a family of h-sets K : Code $\mathcal{U}$ → Type. Importantly, the type of h-sets is itself 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 : Code $\mathcal{U}$ and equivalences e_1 : ⟦ $\mathcal{U} \ni A$ ⟧ ≃ ⟦ $\mathcal{U} \ni B$ ⟧ and e_2 : ⟦ $\mathcal{U} \ni B$ ⟧ ≃ ⟦ $\mathcal{U} \ni C$ ⟧, we can achieve this by adding the field:

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

Intuitively, the family of paths InjComp asserts that Inj must preserve composition of equivalences, i.e. by 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. Concretely, this means that for every code A : Code we can construct the following path,

$$\text{InjRefl}\ A : \text{Inj}\ A\ A\ (\text{id}{\simeq}\ ⟦\ \mathcal{U} \ni A\ ⟧) \equiv \text{refl},$$

where id≃ ⟦ $\mathcal{U} \ni A$ ⟧ : ⟦ $\mathcal{U} \ni A$ ⟧ ≃ ⟦ $\mathcal{U} \ni A$ ⟧ is the identity equivalence on ⟦ $\mathcal{U} \ni A$ ⟧.

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. With this composition, we have constructed a path between $\mathsf{Inj}\ A\ A\ (\mathsf{id}{\simeq}\ \llbracket\ \mathcal{U} \ni A\ \rrbracket)$ 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 $p \equiv \mathsf{refl}$. Hence, by applying this rule we can construct a path between $\mathsf{Inj}\ A\ A\ (\mathsf{id}{\simeq}\ \llbracket\ \mathcal{U} \ni A\ \rrbracket)$ and refl. Furthermore, for any types $X\ Y$: Type, if we let $\mathsf{pathToEquiv}\ X\ Y : X \equiv Y \to X \simeq Y$ be the canonical map from paths to equivalences then by applying the J-elimination rule to the composition of the path witnessing that $\mathsf{pathToEquiv}$ maps reflection to the identity equivalence and the path $\mathsf{InjRefl}\ A$, we can construct the following family of paths:

$$\mathsf{InjSec}\ A\ B\ p : \mathsf{Inj}\ A\ B\ (\mathsf{pathToEquiv}\ (\mathsf{cong}\ \llbracket\ \mathcal{U} \ni\_\rrbracket\ p)) \equiv p$$

In particular, InjSec witnesses that Inj has all sections given by the composition of the functorial action of the interpretation map $\llbracket\ \mathcal{U} \ni\_\rrbracket$ 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 heterogenous over Inj applied to the following canonical equivalences, which correspond to left/right identity and associativity,

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

for all A : $\mathsf{Code}\ \mathcal{U}$, B : $\llbracket\ \mathcal{U} \ni A\ \rrbracket \to \mathsf{Code}\ \mathcal{U}$ and C : $(a : \llbracket\ \mathcal{U} \ni A\ \rrbracket) \to \llbracket\ \mathcal{U} \ni B\ a\ \rrbracket \to \mathsf{Code}\ \mathcal{U}$. 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 $\llbracket \hat{\textstyle\sum} \rrbracket$. To construct $\mathcal{U}$-operadic structures, we require that each of the above equivalences are represented by the paths they are sent to by Inj. Concretely, this means that we require paths between the above equivalences and the application of the map that sends each $X\ Y$: $\mathsf{Code}\ \mathcal{U}$ and $e : \llbracket\ \mathcal{U} \ni A\ \rrbracket \simeq \llbracket\ \mathcal{U} \ni A\ \rrbracket$ to $\mathsf{pathToEquiv}\ (\mathsf{cong}\ \llbracket\ \mathcal{U} \ni\_\rrbracket\ (\mathsf{Inj}\ X\ Y\ e))$ to the same equivalences. Equivalently, given the inverse map to $\mathsf{pathToEquiv}$ following from univalence, i.e. $\mathsf{ua} : X \simeq Y \to X \equiv Y$, we can capture the required conditions with the following fields:

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

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, given careful consideration of size issues, 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.

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. That is, for every collection of operations K : $\mathsf{Code}\ \mathcal{U} \to \mathsf{Type}$, where K is a family of h-sets, we can internalise the notion of a generalised operadic structure on K as a term of a parameterised record type $\mathsf{Operad}\ \mathcal{U}\ K$: Type. Given such an operadic structure O : $\mathsf{Operad}\ \mathcal{U}\ K$, its fields are almost identical to those detailed in Sections 4 and 5, with finite summation or dependent sum over finite sets replaced by $\hat{\textstyle\sum}$. For example, for every A : $\mathsf{Code}\ \mathcal{U}$ and $B : \llbracket\ \mathcal{U} \ni A\ \rrbracket \to \mathsf{Code}\ \mathcal{U}$ the composition map for the operad O is given by the following field:

$$\mathsf{comp}\ O\ A\ B : K\ A \to ((a : \llbracket\ \mathcal{U} \ni A\ \rrbracket) \to K\ (B\ a)) \to K\ (\hat{\textstyle\sum}\ A\ B)$$

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

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

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 $K : \mathcal{U} \to \mathsf{Type}$, where K is a family of h-sets and is equipped with an operad structure O : $\mathsf{Operad}\ \mathcal{U}\ K$, 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 $K\ L\ :\ \mathcal{U}\ \to\ \mathsf{Type}$, where K and L are families of h-sets with operad structures O^K : $\mathsf{Operad}\ \mathcal{U}\ K$ and O^L : $\mathsf{Operad}\ \mathcal{U}\ L$, and given any family of paths $K{\equiv}L : (A : \mathsf{Code}\ \mathcal{U}) \to K\ A \equiv L\ A$, the heterogenous path space between O^K and O^L is equivalent to a pair of heterogenous 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, i.e. the path space between two operads is an h-proposition. The heterogeneous path space between two operads presents a groupoid structure on the type $\sum[\ K \in\ (\mathsf{Code}\ \mathcal{U} \to \mathsf{hSet})\ ]\ \mathsf{Operad}\ \mathcal{U}\ K$ of all $\mathcal{U}$-operads. The full groupoid structure of this construction follows in a natural way from the group structure on path spaces.

We conclude with a brief discussion of 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 will allow 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 10, even the free operad construction can be parameterised over the universe of codes. This highlights one of the key benefits 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.

8 CATEGORY OF OPERADS

To work with operads in HoTT, we require notions for both structure-preserving interactions between them and concrete interpretations of the abstract collection of composable operations that operads represent. In particular, this corresponds to operad homomorphisms and operad algebras. A concrete example of both an operad homomorphism and algebra 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 notion of an 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. Given operads (K, O^K) and (L, O^L), we write $O^K \Rightarrow O^L$ for the type of morphisms between them. This can be expressed in Cubical Agda as a record type where for each $f : O^K \Rightarrow O^L$ we have 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.

In a similar manner to the path space between operads, this type can be seen as a subtype of the type of slice morphisms between K and L, i.e. the type $(A : \mathsf{Code}\ \mathcal{U}) \to K\ A \to L\ A$. That is, the identity and composition laws are h-propositions and hence the path space of operad morphisms is equivalent to the path space of the underlying slice morphisms. Intuitively, two operad morphisms are equal if they act identically on operations. Concretely, this means that given two 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.

The composition of operad morphisms is given by composition of 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 $\langle\ g \bullet f\ \rangle : O^J \Rightarrow O^K$ 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 an operad $O : \mathsf{Operad}\ \mathcal{U}\ K$ is given by defining $\langle\ O\ \rangle\ A\ k = k$, with both the proof of identity and composition preservation holding definitionally. To prove that composition of operad morphisms is associative and unital, we recall that the path space of operad morphisms is equivalent to the path space of their underlying operations. Hence, associativity and identity laws trivially follow from the associativity and unitality of composition of morphisms of slices.

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 a particular *endomorphism operad*, an operad whose operations are 'n-ary like' functions on an h-set and whose composition map is function composition. Concretely, for any h-set $X : \mathsf{Type}$ we begin by defining a collection of operations $\mathsf{EndoOps} : \mathsf{Code}\ \mathcal{U} \to \mathsf{Type}$ that maps each $A : \mathsf{Code}\ \mathcal{U}$ to $([\![\ \mathcal{U} \ni A\ ]\!] \to X) \to X$. Notably, when specialising to the universe Fin, this corresponds to the family of n-ary functions on X. Given that X must be a h-set, it follows that EndoOps is a family of h-sets. We denote the endomorphism $\mathcal{U}$-operad on X by $\mathsf{Endo}\ \mathcal{U}\ X : \mathsf{Operad}\ \mathcal{U}\ \mathsf{EndoOps}$ and its unit operation $\mathsf{id}\ (\mathsf{Endo}\ \mathcal{U}\ X) : ([\![\ \mathcal{U} \ni \hat{\top}\ ]\!] \to X) \to X$ is given by:

```
id (Endo 𝓤 X) f = f (inv (⟦⊤̂⟧ 𝓤) tt)
```

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

```
comp (Endo 𝓤 X) A B f fs : (⟦ 𝓤 ∋ Σ̂ A B ⟧ → X) → X
comp (Endo 𝓤 X) A B f fs xs =
  f (λ a → fs a (λ b → xs (inv (⟦Σ̂⟧ 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,

```
ua→→ f e : PathP (λ i → (ua e i → Y) → Z) f (λ ys → f (ys ∘ fun e))
ua→→ f e i ys = f (ys ∘ ua-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 $[\![\hat{\sum}\mathsf{Idl}]\!]$, $[\![\hat{\sum}\mathsf{Idr}]\!]$ and $[\![\hat{\sum}\mathsf{Assoc}]\!]$ respectively.

9 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 𝓤 : Universe and collection of operations *K* : Code 𝓤 → Type, where *K* is a family of h-sets, the monad over any 𝓤-operad can be internalised in Cubical Agda by first introducing 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 𝓤 is an h-groupoid, then for any operad *O* : Operad 𝓤 *K* the family OpM *O* : Type → Type is equipped with a (strict) 2-monadic structure on the 2-category of h-groupoids, where 1-morphisms are functions and 2-morphisms are paths between such functions. In particular, this monadic structure is simply referred to 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. To define this in HoTT, we introduce a dependent variation of the monadic composition map whereby for every term *ox* : OpM *O X* and function (⟦ 𝓤 ∋ Index *ox*⟧ → *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) = Σ̂ 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 (⟦Σ̂⟧ 𝓤 (Index ox) (Index ∘ next ox f)) k
    in Data (f i (Data ox i)) j
```

From this definition of compM, we can then define the usual monad multiplication map, i.e. join : OpM (OpM *O X*) → *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 Σ̂ 𝓤 (Index *o*) (λ *i* → ⊤̂ 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 2-functorial laws hold definitionally, as required. The proofs that return and join are strict 2-natural transformations similarly hold by definition.

To describe the 2-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* → ⟦ 𝓤 ∋ *p i* ⟧ → *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*).

We proceed by giving an overview for the constructions of the paths witnessing the monadic laws; the details can be found in our formalisation. For all *x* : OpM *O X* we require the two identity laws join-return$_1$ *x* : join (return *x*) ≡ *x* and join-return$_2$ *x* : join (return <$> *x*) ≡ *x*, and for all *y* : OpM *O* (OpM *O* (OpM *O X*)) an associativity law join-assoc : join (join *y*) ≡ join (join <$> *y*). We first construct the required paths on indices as Σ̂Idl 𝓤 (Index *x*), Σ̂Idr 𝓤 (Index *x*) and Σ̂Assoc 𝓤 (Index *y*) (Index ∘ Data *y*) (λ *a* → Index ∘ Data (Data *y a*)). The heterogeneous paths on operations are then given by idl *O* (Index *x*) (Op *x*), idr *O* (Index *x*) (Op *x*) and assoc *O* (Op *y*) (Op ∘ Data *y*) (λ *a* → Op ∘ Data (Data *y a*)). Finally, to construct the corresponding heterogeneous paths on data, we first substitute in the family (λ *p* → PathP (λ *i* → *p i* → *X*)) over the paths ⟦Σ̂Idl⟧ 𝓤 (Index *x*), ⟦Σ̂Idr⟧ (Index *x*) and ⟦Σ̂Assoc⟧ 𝓤 (Index *y*) (Index ∘ Data *y*) (λ *a* → Index ∘ Data (Data *y a*)). From our earlier characterisation of heterogeneous paths of the form PathP (λ *i* → ua *e i* → *X*) *f g*, it suffices to construct the corresponding families of paths of the form (*x* : *X*) → *f x* ≡ *g* (fun *e x*) for each monadic law. The identity laws hold definitionally, while associativity is proved in our formalisation.

For any operad *O* : Operad 𝓤 *K*, the algebras over the strict 2-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. In particular, an operad algebra over *O* is given by a carrier h-set *A* : Type together with an operad morphism *α* : *O* ⇒ Endo 𝓤 *A*. Examples of operad algebras include both the function ⟦_⟧ : IExpr *n* → (Fin *n* → Expr) → Expr with carrier Expr and the function fill : PartialList *A n* → (Fin *n* → List *A*) → List *A* with carrier List *A*. Moreover, this notion of an operad algebra gives rise to an interpretation function runAlg : (*O* ⇒ Endo 𝓤 *A*) → OpM *O A* → *A* defined simply as runAlg *α o* = ⟨ *α* ⟩ (Index *o*) (Op *o*) (Data *o*).

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 ℕ̂ : Universe be the generalised operad universe of totally ordered finite sets, i.e. Code ℕ̂ = ℕ and ⟦ ℕ̂ ∋ *n* ⟧ = Fin *n*, and PList *A* : Operad ℕ̂ (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 moreover preserve the original list with holes in place of the selected 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 𝓤 (List A) be the operad morphism with ⟨ fill⇒ ⟩ = fill, then for every *p* : *A* → 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.

10 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* : ℕ → 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* → ℕ that counts leaves in a rose tree. The fibers of this function, i.e. the countable family of types mapping *n* : ℕ to the type ∑[*t* ∈ RoseTree *A*] count *t* ≡ *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* : ℕ → Type. In particular, a node with *n*-inputs is labelled with an element of *K n* : Type. This is known as the *free planar operad*, and can be defined as follows:

```
data FreePLOps (K : ℕ → Type) : ℕ → Type where
  unit : FreePLOps K 1
  comp : (n : ℕ) (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 𝓤 : Universe, a first attempt at defining the operations of the free 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 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 𝓤 → Type and every code *A* : Code 𝓤 leads to a contradiction. We begin by defining a family of types Partition : Code 𝓤 → Type as follows:

```
Partition X = ∑[ A ∈ Code 𝓤 ] ∑[ B ∈ (⟦ 𝓤 ∋ A ⟧ → Code 𝓤) ]
                 (X ≡ ∑̂ 𝓤 A B)
```

We then construct a family of types FOps : (Code 𝓤 → Type) → Code 𝓤 → Type that is equivalent to the family FreeOps by:

```
FOps K X = (X ≡ ↑̂ 𝓤) ⊎
   (∑[ (A , B , p) ∈ Partition X ] K A × ((a : ⟦ 𝓤 ∋ A ⟧) →
     FreeOps K (B a)))
```

The construction for the family of equivalences between FreeOps *K* and FOps *K* is a routine and provided in our formalisation. As equivalences preserve h-levels, it suffices to show that for any family of h-sets *K* : Code 𝓤 → Type and code *A* : Code 𝓤, the assumption that FOps *K A* is an *h*-set leads to a contradiction. To do this, we consider the case where 𝓤 is the universe of Bishop-finite sets, i.e. Code 𝓤 = FinSet and ⟦ 𝓤 ∋ *A* ⟧ = ⟦ *A* ⟧, *K* is the constant family to the unit type ⊤, and *A* is the finite set size 0 : FinSet. We proceed by recalling that retractions preserve h-levels and therefore it suffices to find a type *X* that is not a h-set and such that we can construct a retraction from FOps (λ *a* → ⊤) (size 0) to *X*. To do this, we can choose *X* = FinSet and then construct a function from FinSet to FOps (λ *a* → ⊤) (size 0) by mapping each *A* : FinSet to

```
inr (A , (λ a → size 0) , ∑̂0 A , tt , λ a → node (size 0) ⊥→ tt ⊥→)
```

where ⊥→ is ⊥-elimination and Σ̂0 *A* : size 0 ≡ Σ̂ *A* (λ *a* → size 0) is constructed by application of the un path constructor.

In the other direction, we can construct a function by induction on each of the left and right cases of FOps (λ *a* → ⊤) (size 0). For the left case we have a proof of size 0 ≡ size 1, which leads to contradiction. For the right case, we simply project out the first component of the partition. In turn, the family of paths that witnesses this function is a retraction follows definitionally, and we can conclude that neither FOps or FreeOps are families of h-sets.

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 𝓤 → Type and *A* : Code 𝓤 to ∥ 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 choosing to present the operations of the free operad on *K* as a higher-inductive family FreeOps *K*, the definition of the operad composition map requires explicit substitutions along paths on codes. In practice, this can lead to "transport hell", whereby operations of the free operad that are constructed by composition are built from 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 presenting an alternative encoding of the operations of the free operad that separates the shape of a tree from the indexing of its leaves.

Our alternative encoding uses a known equivalence between inductive families and small induction-recursion. 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*, for every *K* : Code 𝓤 → Type, to a small inductive-recursive definition. Concretely, we mutually define an inductive type FreeOpsIR *K* : Type together with a recursive function CodeOp *K* : FreeOpsIR *K* → Code 𝓤 as follows:

```
data FreeOpsIR K : Type where
  leaf : FreeOpsIR K
  node : (A : Code 𝓤) → K A →
       (⟦ 𝓤 ∋ A ⟧ → FreeOpsIR K) → FreeOpsIR K

CodeOp K leaf = ⊤̂ 𝓤
CodeOp K (node A k ts) = Σ̂ 𝓤 A (CodeOp ∘ ts)
```

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 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 𝓤 to the 0-truncated type ∥ Σ[*t* ∈ FreeOpsIR *K*] CodeOp *K t* ≡ *A* ∥$_0$. Instead, we require the addition of an appropriate higher path constructor to the definition of 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* → *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 cong (cong *f*) : *p* ≡ *q* → cong *f p* ≡ 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 establish that the functorial action of CodeOp *K* is an embedding, it suffices to show that cong (cong (CodeOp *K*)) : *p* ≡ *q* → cong (CodeOp *K*) *p* ≡ cong (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 cong (cong (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* ≡ *u*, constructs the following path:

```
set p q : PathP (λ i → CodeOp K (p i) ≡ CodeOp K (q i)) refl refl
```

Interestingly, it is possible to show that the set *p q* path constructor is both a section and retraction of cong (cong (CodeOp *K*)), and therefore establish that the fibers of CodeOp *K* are all h-sets. This is a generalisation of *univalent inductive-recursive* types, whereby a path constructor asserting injectivity of the mutually defined function is sufficient to show it has propositional fibers. 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 composition map, we first introduce the following two required families of paths:

```
Σ̂IdlD 𝓤 X : X (inv (⟦⊤̂⟧ 𝓤) tt) ≡ Σ̂ 𝓤 (⊤̂ 𝓤) X

Σ̂AssocD 𝓤 A B C :
  Σ̂ 𝓤 A (λ a → Σ̂ 𝓤 (B a) λ b → C (inv (⟦Σ̂⟧ 𝓤 A B) (a , b)))
                 ≡ Σ̂ 𝓤 (Σ̂ 𝓤 A B) C
```

where *X* : ⟦ 𝓤 ∋ ⊤̂ ⟧ → Code 𝓤, *A* : Code 𝓤, *B* : ⟦ 𝓤 ∋ *A* ⟧ → Code 𝓤 and *C* : ⟦ 𝓤 ∋ Σ̂ 𝓤 *A B* ⟧ → Code 𝓤. While we do not give the full constructions for these paths here, they follow from the paths Σ̂Idl and Σ̂Assoc. We proceed by defining a variant of the operad composition map whose input operations are not indexed. In particular, for each tree *t* : FreeOpsIR *K* and family of trees *ts* : ⟦ 𝓤 ∋ CodeOp *K t* ⟧ → FreeOpsIR *K* we construct a term

```
graft t ts : fiber (CodeOp K (Σ̂ 𝓤 (CodeOp K) (CodeOp K ∘ ts))
```

as follows:

```
graft leaf ts =
   ts (inv (⟦⊤̂⟧ 𝓤) tt) , Σ̂IdlD 𝓤 (CodeOp K ∘ ts) (inv (⟦⊤̂⟧ 𝓤) tt)
graft (node A k ts) tss
  = let iv = inv (⟦Σ̂⟧ 𝓤 A (CodeOp K ∘ ts))
        us = graft (ts a) λ b → tss (iv (a , b))
        ac = Σ̂AssocD 𝓤 (CodeOp K ∘ ts) (CodeOp K ∘ tss)
    in node A k (fst ∘ us) , (λ i → Σ̂ 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 ≡ $\hat{\top}$ $\mathcal{U}$ holds definitionally. Finally, while we do not provide the constructions for the paths witnessing the identity and associativity operad laws were, they require significant manipulation of nested substitutions and can be found in our formalisation.

We have thus far described the operadic structure on the fibers over the function CodeOp K for every $\mathcal{U}$-species K : Code $\mathcal{U}$ → Type, but have not yet shown this is the *free* operad on K. Indeed to do this, 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 $\mathcal{U}$ → 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 $\mathcal{U}$ → Type, their hom is given by the type of natural transformations (A : Code) → $K\ A \to L\ A$, 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 detailed in Section 8.

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 ⟨_⟩ : $(O^K \Rightarrow O^L) \to$ (A : Code) → $K\ A \to L\ A$. The free operad on a $\mathcal{U}$-species K : Code $\mathcal{U}$ → 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 : Code $\mathcal{U}$ → Type to the collection of operations fiber (CodeOp K) : Code $\mathcal{U}$ → Type equipped with its previously described operadic structure. The functoriality of this construction follows from our proof that it satisfies the universal property of being a left adjoint in the usual way. The first step to showing that this is indeed left adjoint to the forgetful functor from $\mathcal{U}$-operads to $\mathcal{U}$-species, is to define the following unit natural transformation:

```
η : (A : Code 𝓤) → K A → fiber (CodeOp K) A
η A k = node A k (λ a → leaf) , Σ̂Idr 𝓤 A
```

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

```
interpret O f .(⊤̂ 𝓤) leaf = id O
interpret O f .(Σ̂ 𝓤 A B) (node A B k ts) =
   comp O A B (f A k) (λ a → interpret O f (B a) (ts a))
```

As can be observed, interpret definitionally respects the operad identity. The proof that it also respects operad composition and thus extends to a morphism between the operads FreeOperad K and O requires manipulation of nested substitutions and is provided in our library. Moreover, our formalisation necessarily includes a construction for the universal property of the free operad, namely a proof that for all f : (A : Code $\mathcal{U}$) → $K\ A \to L\ A$ the type

```
Σ[ g ∈ FreeOperad K ]
   ((A : Code 𝓤) (k : K A) → ⟨ g ⟩ A (η A k) ≡ f A k)
```

is contractible with the base point given by a pair of the function interpret $O\ f$ extended to an operad map together with a witness of the proposition that for all A : Code $\mathcal{U}$ and $k : K\ A$, interpret $O\ f\ A\ (\eta\ A\ k) = f\ A\ k$.

11 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 [11]. More recently, containers were developed to capture the idea of the strictly positive types [1]. The more general presentation of indexed containers was later developed to capture inductive families [4]. Containers are closed under finite products and exponentials [5], and have a sensible 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 [14], namely Bishop-finiteness. However, there has also been work on internalising ‘enumerated’ types and Kuratowski finite sets in the framework of HoTT [9, 10]. 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 [12]. 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 [14]. In particular, Yorgey internalises combinatorial species to capture specific classes of data types, analogous to shapely types and the theory of containers.

The theory of operads originated in the field of algebraic topology [7, 13] for describing operations on iterated loop-spaces.

12 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 show how discretely finite containers, which represent inductive data types whose constructors have a finite arity, have a natural operadic interpretation. Intuitively this provides an operation-centric approach to inductive types, in contrast to the traditional data-centric approach. Our results open up a new line of work on investigating the properties and applications of operads from a type-theoretic perspective. 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. 2003. *Categories of Containers*. Ph.D. Dissertation. University of Leicester.

[2] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. 2005. Containers: Constructing Strictly Positive Types. *Theoretical Computer Science* 342, 1 (2005).

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

[4] Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor McBride, and Peter Morris. 2015. Indexed Containers. *Journal of Functional Programming* 25 (2015).

[5] Thorsten Altenkirch, Paul Levy, and Sam Staton. 2010. Higher-Order Containers. In *Conference on Computability in Europe*. Springer.

[6] Anonymous. 2024. HoTT Operads. https://github.com/Anonymised23/HoTTOperads.

[7] John Michael Boardman and Rainer M Vogt. 2006. *Homotopy Invariant Algebraic Structures on Topological Spaces*. Vol. 347. Springer.

[8] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. 2016. Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom. *arXiv preprint arXiv:1611.02108* (2016).

[9] Thierry Coquand and Arnaud Spiwack. 2010. Constructively Finite?. In *Contribuciones Científicas en Honor de Mirian Andrés Gómez*. Universidad de La Rioja.

[10] Dan Frumin, Herman Geuvers, Léon Gondelman, and Niels van der Weide. 2018. Finite Sets in Homotopy Type Theory. In *International Conference on Certified Programs and Proofs*.

[11] C Barry Jay and J Robin B Cockett. 1994. Shapely Types and Shape Polymorphism. In *European Symposium on Programming*. Springer.

[12] André Joyal. 1981. Une Théorie Combinatoire des Séries Formelles. *Advances in Mathematics* 42, 1 (1981).

[13] J Peter May. 2006. *The Geometry of Iterated Loop Spaces*. Vol. 271. Springer.

[14] Brent Abraham Yorgey. 2014. *Combinatorial Species and Labelled Structures*. Ph.D. Dissertation. University of Pennsylvania.