Part III Dissertation

Generalised Species of Structures in Homotopy Type Theory Using Agda

Rupert Horlick (rh572)
Homerton College

Supervisor: Prof. M. Fiore

University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thompson Avenue
Cambridge, CB3 0FD
United Kingdom

Abstract

Generalised Species of Structures in Homotopy Type Theory Using Agda
The abstract goes here!

Word count is X

Contents

1 Introduction

This project formalises the theory of generalised species of structures [4] embedded in homotopy type theory [11] using Agda [10]. The formalisation is built on the HoTT-Agda library [2], a thorough implementation of homotopy type theory. The basis of the project is an interpretation of the constructs of homotopy type theory as categorical structures. This leads to a neat formalisation of the theory of generalised species within the type theory.

The project has three main parts. First the formalisation of presheaves, a type of categorical functor, along with operations for manipulating them. Second the formalisation of the free strict symmetric-monoidal completion that generalises the set-theoretic finite-multiset construction. This construction was the most significant challenge of the project and we ended up having two implementations of it. Lastly, with the first two parts, the formalisation of generalised species themselves. This includes the differential calculus of species, a set of operations and laws for combination and manipulation of species. This culminates in the species version of the Leibniz rule for differentiation.

1.1 Related Work

The study of species began with the work of Joyal [6, 7] who gave the original definition and showed their relation to exponential power series. This work was explored further by Bergeron et al. [1] in an extensive volume. The notion of species was expanded to that of generalised species of structures by Fiore et al. [4] who showed the Cartesian Closure of the bicategory of generalised species and showed the links between species and models of linear logic. Further Fiore [3] defined the differential calculus of species.

Homotopy type theory began in 2012-13 during a Special Year on Univalent Foundations of Mathematics at the Institute of Advanced Study. The Special Year led to the official book [11], a thorough look at homotopy type theory and its current applications. The PhD thesis of Yorgey [12] defines Joyal's species in the context of homotopy type theory, but was not formalised.

The formalisation of a theory of groupoids in homotopy type theory requires higher structure. The work of Hou [5] looks specifically at the formalisation of such structures in Agda. The work of Kraus [8] generalises some useful notions form the former that apply when defining actions on groupoids.

2 Background

2.1 Category Theory

We assume that the reader has a grasp of the basic notions of category theory. That is we assume familiarity with categories, functors, natural transformations, initial/terminal objects, products/coproducts, limits/colimits in general, exponentials, Cartesian Closed Categories, and the Yoneda Lemma.

Two notions that do not usually appear in introductory courses are groupoids and ends/coends. These will both be required throughout this dissertation, so their definitions will be discussed here.

Groupoids A groupoid is a special kind of category, specifically one where every morphism has an inverse. This property induces an isomorphism between $\mathbb{G}$ and $\mathbb{G}^{\mathrm{op}}$ for every groupoid $\mathbb{G}$.

Ends/Coends Ends/coends are a type of limit/colimit over functors of the form $F : \mathbb{C}^{\mathrm{op}} \times \mathbb{C} \to \mathbb{D}$. For such a functor, a wedge $w : e \to F$ is an object $w : \mathbb{D}$ and a family of morphisms $e_c : w \to F(c, c)$, for each $c \in \mathbb{C}$, such that, for every morphism $f : c \to c'$ in $\mathbb{C}$, the diagram

$$\begin{tikzcd}
w \arrow[r, "e_{c'}"] \arrow[d, "e_c"'] & F(c', c') \arrow[d, "F(f, c')"] \\
F(c, c) \arrow[r, "F(c, f)"] & F(c, c')
\end{tikzcd}$$

commutes. We can compose with a map $g : v \to w$ to form a new wedge $e\ f : v \to F$. An end is a universal wedge, i.e. a wedge $e : w \to F$ such that for any other wedge $e' : w' \to F$, the latter factors through the former by a unique map $w' \to w$. The dual notion of cowedge leads to the definition of coends.

2.2 Type Theory

Homotopy type theory is a new area of mathematics that combines homotopy theory and Martin-Löf type theory [9]. Homotopy theory deals with *spaces* and *continuous maps* between spaces, up to homotopy. A *homotopy* between two continuous maps is a "continuous" deformation of one map into the other. Two spaces X and Y are *homotopy equivalent*, $X \simeq Y$, if we have continuous maps in both directions, whose compositions both have homotopies to the identity maps.

Homotopy type theory uses these spaces as the interpretation of type theory. The statement "a has type A" or $a : A$, is taken to mean "a is a point in the space A". Type constructions, such as functions and product, can be seen as homotopy-invariant constructions on spaces.

A Quick Note on Equality To talk about homotopy type theory we need several notions of equality. *Definitional equality*, denoted $:\equiv$, allows us to make two terms equal by definition. *Judgemental equality*, denoted $\equiv$, means that two terms compute to the same normal form based on existing definitions. Finally *propositional equality*, denoted $=$, is internal to type theory and will be defined below.

Type Formers To make type theory useful we need the ability to build new types. Defining a type means specifying how to form values of the type, how to eliminate values of the type, and how to compute with values of the type. These specifications are called introduction rules, elimination rules, and computation rules.

Many common types fall under the label of *inductively defined* types. These consist of a number of constructors with optional arguments; an application of each constructor is an introduction rule. The elimination and computation rules can also be derived from the constructors. To eliminate a value of an inductively defined type, one must perform case analysis on the value and provide a result for each case. Given a specific value of the type, computation picks the correct result based on the case analysis.

Some types are not defined inductively. These types do not use the framework of constructors given above, and have to define their own introduction, elimination, and computation rules. Important examples of this are the basic function and product types, as well as their dependent versions, Π and Σ types.

Functions are formed using λ-abstraction and eliminated using function application. The computation rule says

$$(\lambda x.C(x))(a) \mapsto C(a).$$

Dependent functions or Π types have the same rules, except that now the type of the result of the function may depend on the value passed into the function. The type of dependent functions from A to $B : A \to \mathcal{U}$ is written $\Pi_{(a:A)} B(a)$.

For any other type, we can package the elimination and computation rules into an elimination principle. This tells us how we can form a function from the type to some family C that may depend on it. For example, for the type of dependent pairs, $\Sigma_{(a:A)} B(a)$, we have

$$\mathrm{elim}_{\Sigma_{(a:A)} B(a)} : \Pi_{(C:\Sigma_{(a:A)} B(a)\to\mathcal{U})}(\Pi_{(a:A)}\Pi_{b:B(a)} C(a,b)) \to \Pi_{(p:\Sigma_{(a:A)} B(a))} C(p)$$

which is defined by

$$\mathrm{elim}_{\Sigma_{(a:A)} B(a)}(C, g, (a,b)) :\equiv g(a)(b).$$

So to define a function on the dependent product it is enough to define a function on the components of the product. In the case that C does not depend on its argument we get the simpler recursion principle, which for Σ types has the type

$$\mathrm{rec}_{\Sigma_{(a:A)} B(a)} : \Pi_{(C:\mathcal{U})}(\Pi_{(a:A)} B(a) \to C) \to \Sigma_{(a:A)} B(a) \to C$$

and the same definition as before.

Identity Type The interpretation of types as spaces extends to the notion of propositional identity, $a =_A b$, of elements a and b of the same type A, treating it as the space of a paths $p : a \rightsquigarrow b$ from a to b in the space A. Terms $p : a =_A b$ of the identity type on A are now exactly these paths.

The path space is defined using an inductive type with one constructor,

$$\mathrm{idp} : \Pi_{(a:A)} a = a.$$

We denote this path for an element $a : A$ as idp_a. There is also a notion of path composition, denoted, for two paths $p : a =_A b$ and $q : b =_A c$, $p \cdot q$. Note that the direction of composition is diagrammatical and therefore opposite to the more usual $q \cdot p$. Every path has an inverse, denoted $!p$. The identity path is its own inverse and any path composed with its own inverse gives the identity path, i.e.

$$p \cdot !p = \mathrm{idp}_a$$

and

$$!p \cdot p = \mathrm{idp}_b.$$

As the path space is an inductive type, it comes equipped with the introduction, elimination, and computation rules of an inductive type. The elimination rule is a case analysis on the constructors, but of course there is only one constructor. However, this constructor is only defined when both sides of the equality are exactly the same value, thus to perform the case analysis we must assume that this is the case. This leads to the elimination principle, which we refer to as path induction, that has the type

$$\mathrm{elim}_{=_A} : \Pi_{(C:\Pi_{(x,y:A)} (x=y)\to\mathcal{U})}(\Pi_{(x:A)} C(x,x,\mathrm{idp}_x)) \to \Pi_{(x,y:A)}\Pi_{(p:x=y)} C(x,y,p)$$

and defining equation

$$\mathrm{elim}_{=_A}(C, c, x, x, \mathrm{idp}_x) :\equiv c(x).$$

This says that to prove something about a path $p : x = y$, it is enough to show the property when p is idp_x and y is x.

This principle is a very powerful reasoning tool and it allows us to define functions in Agda on paths by pattern matching on idp. So for example we can define the useful function

$$\mathrm{ap} : (f : A \to B) \to x = y \to f\ x = f\ y$$

as

$$\mathrm{ap}\ f\ \mathrm{idp} :\equiv \mathrm{idp}\,.$$

When the pattern match is made, the result type becomes $f\ x = f\ x$ which is inhabited by $\mathrm{idp}_{f\ x}$.

We can also define dependent paths. For a dependent type B over A, a path $p : x = y$ in A, and two points $u : B\ x$ and $v : B\ y$, the type of dependent paths is the type of paths from u to v lying over p, denoted $u = v\ [B \downarrow p]$. In the case that p is idp this reduces to $u = v$, and by path induction we use this as the definition.

Univalence In type theory there are universes, $\mathcal{U}$, the elements of which are themselves types. There is an identity type $\mathrm{Id}_{\mathcal{U}}$, which relates the types, that is again interpreted as the space of paths $p : A \rightsquigarrow B$ in the space $\mathcal{U}$. The *univalence axiom* states that such paths correspond exactly to homotopy equivalences, i.e.

$$(A = B) \simeq (A \simeq B),$$

which allows us to identify isomorphic types. This may be seen as formalising the common practice of working "up to isomorphism". The righthand direction of this equivalence already holds, because we can extract the equivalence defined by a path. Using path induction, for idp_A we have the function $\mathrm{id}_A : A \to A$ which is clearly an equivalence. The lefthand direction does not necessarily hold, so this is the part that is axiomatised.

Truncation Levels Every type has a path space defined by the identity type. These path spaces themselves have identity types, so there is a hierarchy of path structure for every type. The *truncation level* of a type describes the point in the hierarchy at which the path structure becomes trivial. Intuitively a type can be seen as trivial if it corresponds to a singleton type, which is captured by the notion of contractibility, defined by

$$\text{is-contr}\ A :\equiv \Sigma_{(x:A)}\, \Pi_{(y:A)}\ x = y.$$

The type has some base point x to which all other points are equal, so it can be to contain only a single value. Type which are contractible are said to have truncation level -2 for historical reasons.

The next level up, level -1, is the level of propositions, which are types whose path spaces are contractible, i.e. types satisfying

$$\text{is-prop}\, A :\equiv \Pi_{(x,y:A)}\ \text{is-contr}\,(x = y).$$

This definition is equivalent to

$$\text{has-all-paths}\, A :\equiv \Pi_{(x,y:A)}\ x = y,$$

so types at this level have all elements equal, but they may not be inhabited. This acts like a proposition because inhabitance intuitively corresponds to constructive truth, with every proof of inhabitance being identical.

Another level up, level 0, we have types whose paths spaces are propositions. These types can be viewed as sets by taking equivalence classes, i.e. collections of elements that are propositionally equal, as elements. A set should not have any more structure than this, so it makes sense for any paths between paths to be equal.

An important level for this dissertation is level 1, which is the level of groupoids. Types with this level satisfy

$$\text{is-gpd}\, A :\equiv \Pi_{(x,y:A)}\ \text{is-set}\,(x = y),$$

and can be seen to be groupoid-like, corresponding to the definition in Section 2.1, because they have a set of invertible paths between elements of a type, corresponding to the homs of a groupoid. This view is fundamental to the work in this dissertation and is elaborated in the next chapter.

This tower of levels carries on up to infinity, so we can recursively define what it means for a type to have level $\mathrm{S}\, n$, where S is the successor on truncation levels, as

$$\text{has-level}\,(\mathrm{S}\, n)\ A :\equiv \Pi_{(x,y:A)}\ \text{has-level}\, n\,(x = y).$$

The base case of this is at -2, where the definition is the same as that of is-contr.

It is useful to refer to sub-universes, which contain only types of a given level. For a given level n this can be denoted $\mathcal{U}_n$. For levels -1, 0, and 1, these will be referred to as **hProp**, **hSet**, and **hGpd** respectively. The definition of these is

$$\mathcal{U}_n :\equiv \Sigma_{(A:\mathcal{U})}\ \text{has-level}\, n\ A.$$

This definition leads to the projection $\pi_1 : \mathcal{U}_n \to \mathcal{U}$ sometimes appearing in type signatures to extract the underlying type.

Truncation It is useful to be able to force a type to have a particular truncation level and this is achieved using the truncation operation. For a truncation level n the n^{th} truncation of A is denoted

$$\|A\|_n.$$

The definition of this operation is omitted.

3 Categorical Interpretation

The starting point for this project is an interpretation of the existing constructs of homotopy type theory as categorical structures.

The category Set The category **Set** has sets as objects and functions between sets as morphisms. This informal definition depends on the mathematician and the formal foundations they use.

At first sight, the universe, $\mathcal{U}$, could be interpreted as **Set**. Now types would be interpreted as sets and functions between types as functions between sets. Composition, identities, and associativity all hold for functions between types.

In category theory **Set** is a Cartesian Closed Category, i.e. it has a terminal object, products, and exponentials. $\mathcal{U}$ also has this structure, with the unit type, $\top$, being the terminal object, product types being categorical products, and function spaces being categorical exponentials.

Without going into too much detail at this point, using $\mathcal{U}$ as **Set** does not work for our purposes. To form more complex categorical constructions in the rest of this dissertation we will need to reason about the truncation level $\mathcal{U}$, but of course the universe is not truncated.

In section 2.2 we saw that types with truncation level 0 can be viewed as sets and also that there is a sub-universe **hSet** containing all such types. This sub-universe can be interpreted as the category **Set**, and allows reasoning about its truncation level.

Types as Groupoids Types themselves have a set of objects, their elements, and in homotopy type theory come equipped with a natural intrinsic morphism structure, namely the path space. So for a given type, C, and two elements, c and c', the hom $C(c, c')$ becomes $c = c'$. All paths are invertible, so (C, Id_C) has a groupoid structure. Identity morphisms are given by identity paths and composition of morphisms by path composition. The unit and associativity laws come from the same laws for paths.

The 2-category Gpd Viewing (C, Id_C) as a groupoid for any type naively leads to an alternative interpretation of the universe as the 2-category **Gpd**. The objects are exactly these pairs of types and corresponding identity types. The morphisms are functors between groupoids. Functions, as well as being morphisms in **Set**, can be treated as functors. Their action on objects is just function application and their action on morphisms, here paths, is given by the function $\mathrm{ap} : (f : C \to D) \to x = y \to f\ x = f\ y$, defined in Section 2.2. All functions $f : C \to D$ are functorial because of some basic facts of homotopy type theory, i.e. functions preserve identities

$$\mathrm{ap}\ f\ \mathrm{idp} = \mathrm{idp},$$

and functions preserve composition

$$\mathrm{ap}\ f\ (p \cdot q) = \mathrm{ap}\ f\ p \cdot \mathrm{ap}\ f\ q.$$

In category theory, the 2-cells of **Gpd** are natural transformations. Recall that for two functors, $F, G : \mathbb{G} \to \mathbb{H}$, a natural transformation, $\varphi : F \Rightarrow G$, is a family of morphisms

$$\varphi_x : F\ x \to G\ x,$$

for each object $x \in \mathbb{G}$, such that for any $f : x \to y$

$$\begin{tikzcd} F\ x \arrow[r, "\varphi_x"] \arrow[d, "F f"'] & G\ x \arrow[d, "G f"] \\ F\ y \arrow[r, "\varphi_y"] & G\ y \end{tikzcd}$$

commutes.

For two functions $f, g : C \to D$ regarded as functions between the associated groupoids, this becomes a family of paths

$$\varphi_x : f\ x =_D g\ x.$$

The naturality condition of the above diagram becomes the condition that, for any path $p : x = y$,

$$(\mathrm{ap}\ f\ p) \cdot \varphi_y = \varphi_x \cdot (\mathrm{ap}\ g\ p)$$

and one can show that it holds for all

$$\varphi : \Pi_{(x:C)}\ f\ x =_D g\ x.$$

Indeed, by path induction, assume p is idp and y is x. The path now reduces to

$$\varphi_x = \varphi_x$$

which is inhabited by idp.

Other basic type formers can be viewed as categorical constructions in this 2-category **Gpd**. The product of types becomes the product of groupoids, with morphisms taken pointwise. The coproduct of types becomes the coproduct of groupoids. The path space here is only non-empty when both values are inl or inr and in this case matches the underlying one. Π types/Σ types become ends/coends, defined in Section 2.1.

4 Presheaves

4.1 $\mathcal{U}$-Presheaves

In category theory, a presheaf on a category $\mathbb{C}$ is a functor $\mathbb{C}^{\text{op}} \to \mathbf{Set}$. As for a groupoid $\mathbb{G} \cong \mathbb{G}^{\text{op}}$, without loss of generality, under our interpretation, $\mathbb{C}^{\text{op}}$ becomes (C, Id_C) and $\mathbf{Set}$ can be taken, in the first instance, to be $\mathcal{U}$. $\mathcal{U}$ is being treated as a category here, so the definition of functors between groupoids from Chapter 3 does not apply.

Instead, we want a mapping on objects

$$c : C \mapsto F\ c : \mathcal{U}$$

and a mapping on morphisms

$$(c = c') \mapsto (F\ c \to F\ c').$$

In homotopy type theory we can define the function

$$\text{coe} : (c = c') \to c \to c'$$

using path induction as

$$\text{coe idp}\ c :\equiv c.$$

In combination with ap this gives

$$\text{coe} \circ \text{ap}\, F : (c = c') \to F\ c \to F\ c'$$

as we need. This is such a common operation in homotopy type theory that it has its own name, transport. Thus we consider functions $F : C \to \mathcal{U}$ as presheaves on C and will write the type of presheaves on C as $\widehat{C}$.

A natural transformation, $\theta : \text{Nat}(P, Q)$, between two presheaves, $P, Q : \mathbb{C} \to \mathbb{D}$, is a family of morphisms,

$$\theta_x : P\ x \to Q\ x,$$

such that for all $x, y \in \mathbb{C}$ and $f : x \to y$ the following diagram commutes.

$$\begin{tikzcd}
P\ x \arrow[r, "\theta_x"] \arrow[d, "P f"'] & Q\ x \arrow[d, "Q f"] \\
P\ y \arrow[r, "\theta_y"] & Q\ y
\end{tikzcd}$$

For $P, Q : \widehat{C}$, this translates directly to the Π type

$$\mathrm{Nat}(P, Q) :\equiv \prod_{(c:C)} P\ c \to Q\ c.$$

For an element of this type to be natural it needs to satisfy, for all $p : x = y$,

$$\theta_y \circ \text{ transport } P\ p = \text{transport } Q\ p\ \circ\ \theta_x.$$

This holds for all elements of the type Nat by path induction as, by the definition of transport, we have

$$\text{transport } P\ \text{ idp} = \text{id},$$

where id is the identity function and by the left and right unit laws of function composition we have to show $\theta_x = \theta_x$ which is inhabited by idp.

The category of presheaves is a Cartesian Closed Category (CCC), which means that it has a terminal object, products, and exponentials. The terminal object is the constant function on the unit type, ⊤. The product of two presheaves $P, Q : \widehat{C}$ is taken pointwise, so

$$(P \times Q)\ c :\equiv P\ c \times Q\ c.$$

The exponential object for two presheaves is defined as

$$(P \Rightarrow Q)\ c :\equiv (Y\ c \times P) \Rightarrow Q.$$

4.2 Yoneda

The Yoneda functor, $Y : \mathbb{C} \to \widehat{\mathbb{C}}$, takes each object of $\mathbb{C}$ to the presheaf

$$Y\ c :\equiv \mathbb{C}(c, -),$$

i.e. to the presheaf that for all $x \in \mathbb{C}$ returns the hom from c to x. Our interpretation takes homs to identity types, which means that the Yoneda functor becomes

$$Y\ c\ x :\equiv c = x.$$

The Yoneda Lemma states that given an object $c \in \mathbb{C}$, the set of natural transformations from $Y\ c$ to some other presheaf X is isomorphic to the set $X\ c$, which in type theory becomes

$$\mathrm{Nat}(Y\ c, P) \simeq P\ c.$$

Let us look at the Agda proof of the Yoneda Lemma to get an idea of the steps involved in formalisation.

```
yonedaLemma : ∀ (P : prshf i C) (c : C)
            → P c ≃ nat (yon c) P
yonedaLemma P c = equiv f g (λ b → λ= (λ x → λ= (f-g b x))) (λ _ → idp)

  where

    f : P c → nat (yon c) P
    f p x idp = p

    g : nat (yon c) P → P c
    g p = p c idp

    f-g : (b : nat (yon c) P) → (x : C) → (p : yon c x)
      → f (g b) x p == b x p
    f-g b x idp = idp
```

The proof looks like what one might write on paper. To show that this is an equivalence, it is enough to exhibit functions in each direction, `f` and `g`, and show that their compositions act like the respective identities.

Density Formula The density formula is related to the Yoneda Lemma and will be used in several proofs. It is defined as

$$P\ c \simeq \Sigma_{(x:C)}\ (P\ x) \times (x = c)$$

The Agda proof is again very similar to pen and paper, defining an equivalence by giving functions in both directions and showing their compositions act as identities.

```
density : ∀ (P : prshf C) (c : C) → P c ≃ Σ C (λ x → (P x) × (x == c))
density P c = equiv f g f-g (λ _ → idp)

  where

    f : P c → Σ C (λ x → P x × (x == c))
    f p = (c , p , idp)
```

```
g : Σ C (λ x → P x × (x == c)) → P c
g (_ , q , idp) = q

f-g : (b : Σ C (λ x → P x × (x == c))) → f (g b) == b
f-g (_ , q , idp) = idp
```

4.3 Left Kan Extension

Consider the following diagram

$$\begin{tikzcd} C \arrow[r, "Y"] \arrow[dr, "F"'] & \widehat{C} \arrow[d, "\mathrm{Lan}_Y F"] \\ & \widehat{D} \end{tikzcd}$$

The function $\mathrm{Lan}_Y\, F$ is the left Kan extension of the function F along the Yoneda functor. This gives us a way to lift functions to act on presheaves. There is a general definition of the left Kan extension, but for our purposes it is enough to define it just for Yoneda. In this case the definition is

$$\mathrm{Lan}_Y\, F\, P\, d \;:\equiv \Sigma_{(c\,:\,C)}\, P\, c \times F\, c\, d$$

using a coend which here is a Σ type.

Lan_Y has many interesting properties that will be used when proving properties of species. The left Kan extension of the Yoneda functor is the identity function, which is proved by an application of the density formula,

$$\mathrm{Lan}_Y\, Y = \mathrm{id}\,.$$

The Yoneda functor composed with $\mathrm{Lan}_Y\, F$ gives back F,

$$\mathrm{Lan}_Y\, F \circ Y = F,$$

again proved using the density formula. The composition of two Lan_Ys reduces to the Lan_Y of one Lan_Y composed with the first function,

$$\mathrm{Lan}_Y\, G \circ \mathrm{Lan}_Y\, F = \mathrm{Lan}_Y\, (\mathrm{Lan}_Y\, G \circ F),$$

which is proved by a simple permutation of the components of the Σ type.

For any two monoids, C and D, and a monoid homomorphism, $F\, :\, C \to \widehat{D}$, where the monoid of the codomain is given by Day Convolution, we have that $\mathrm{Lan}_Y\, F$ is also a monoid homomorphism with respect to Day Convolution in both the domain and codomain. The unit law is proved using a single application of the density formula and the unit law for F. The multiplication law is proved similarly with an application of the density formula and the multiplication law for F, however this time requires permutations of the components of the Σ type before and after these are applied.

4.4 Day Convolution

Given a category, $\mathbb{C}$, that has a monoidal structure, Day Convolution extends the structure to presheaves on $\mathbb{C}$. The associative operation, $\hat{\otimes}$, is defined using two coends as

$$(P \mathbin{\hat{\otimes}} Q)\ c \ :\equiv \int^{(c_1, c_2 \in \mathbb{C})} P\ c_1 \times Q\ c_2 \times \mathbb{C}(c_1 \otimes c_2, c).$$

Under our interpretation coends are Σ types and homs are path spaces, leading to the definition

$$(P \mathbin{\hat{\otimes}} Q)\ c \ :\equiv \Sigma_{(c_1 : C)} \Sigma_{(c_2 : C)}\ P\ c_1 \times Q\ c_2 \times (c = c_1 \otimes c_2).$$

The identity presheaf for this operation makes reference to the identity for the underlying structure, the identity being defined as

$$\hat{\mathrm{I}} \ :\equiv Y\ e.$$

To form a commutative monoid these need to be accompanied by proofs of the left and right unit laws, the associativity law, and the commutativity law. We will look in detail at the proof of the left unit law, first as a pen-and-paper proof and then in Agda. The left unit law for a monoid states that

$$\Pi_{(c : C)}\ e \otimes c = c$$

which for the Day Convolution becomes

$$\Pi_{(P : \hat{C})}\ \hat{\mathrm{I}} \mathbin{\hat{\otimes}} P = P.$$

By function extensionality and the univalence axiom it is enough to show

$$\Pi_{(P : \hat{C})} \Pi_{(c : C)}\ (\hat{\mathrm{I}} \mathbin{\hat{\otimes}} P)\ c \simeq P\ c.$$

Taking arbitary P and c we have

$$\begin{aligned}
(\hat{\mathrm{I}} \mathbin{\hat{\otimes}} P)\ c &\equiv \Sigma_{(c_1 : C)} \Sigma_{(c_2 : C)}\ (c_1 = e) \times P\ c_2 \times (c = c_1 \otimes c_2) & \\
&\simeq \Sigma_{(c_2 : C)}\ P\ c_2 \times \Sigma_{(c_1 : C)}\ (c = c_1 \otimes c_2) \times (c_1 = e) & \text{(commutativity of } \Sigma\text{)} \\
&\simeq \Sigma_{(c_2 : C)}\ P\ c_2 \times (c = e \otimes c_2) & \text{(density formula)} \\
&\simeq \Sigma_{(c_2 : C)}\ P\ c_2 \times (c = c_2) & \text{(left unit law for } \otimes\text{)} \\
&\simeq \Sigma_{(c_2 : C)}\ P\ c_2 \times (c_2 = c) & \text{(symmetry of paths)} \\
&\simeq P\ c & \text{(density formula)}
\end{aligned}$$

The Agda proof below gives a representative example of 'equational reasoning' proofs in Agda. Equational reasoning proofs chain together several steps of an equivalence or equality and include the types of each step for clarity.

```
⊗̂-unit-l : (P : prshf C) → ∀ c → (Î ⊗̂ P) c ≃ P c
⊗̂-unit-l P c =

  (Î ⊗̂ P) c

    ≃⟨ perm ⟩

  Σ C (λ c₂ → P c₂ × Σ C (λ c₁ → (c == (c₁ ⊗ c₂)) × (c₁ == e)))

    ≃⟨ Σ-emap-r (λ c₂ →
        ×-emap-r
          (P c₂)
          ((density (λ p → c == (p ⊗ c₂)) e)⁻¹)) ⟩

  Σ C (λ c₂ → P c₂ × (c == (e ⊗ c₂)))

    ≃⟨ Σ-emap-r (λ c₂ →
        ×-emap-r
          (P c₂)
          (coe-equiv (ap (c ==_) ⊗-unit-l))) ⟩

  Σ C (λ c₂ → P c₂ × (c == c₂))

    ≃⟨ Σ-emap-r (λ c₂ →
        ×-emap-r
          (P c₂)
          (!-equiv)) ⟩

  Σ C (λ c₂ → P c₂ × (c₂ == c))

    ≃⟨ (density P c)⁻¹ ⟩

  P c

    ≃∎
```

Each step of the proof corresponds to one step of the pen-and-paper proof. The only difference in the formalisation is that the justifications have to specify exactly what context they are being applied in and how exactly they are being applied.

We have a couple of results about monoid homomorphisms with respect to Day Convolution. Firstly the Yoneda functor is a homomorphism, because applied to e it returns $\hat{\mathrm{I}}$ and applied to $x \otimes y$, we can show that we get $Y\,x \mathbin{\hat{\otimes}} Y\,y$.

4.5 $\mathcal{U}_n$-Presheaves

- Discuss Yoneda, Lans, and Day again here

The definition of Day Convolution above is for $\mathcal{U}$-valued presheaves, but as mentioned in ? we will have to restrict to **hSet** or **hGpd**-valued presheaves later on. We cannot naïvely use the previous definition, extracting the underlying type from the **hSet**/**hGpd**, because the resulting type contains elements of C, which are not necessarily **hSet**s/**hGpd**s. We could restrict to the case that C does have the correct level, however this will not work for us. Instead, when we are working with a type C that has level $\mathrm{S}\,(\mathrm{S}\,n)$, we will want to use $\mathcal{U}_{(\mathrm{S}\,n)}$-valued presheaves. This is achieved using truncation as

$$(P \mathbin{\hat{\otimes}} Q)\,c \;:\equiv\; \left\| \Sigma_{(c_1 : C)} \Sigma_{(c_2 : C)}\; P\,c_1 \times Q\,c_2 \times (c = c_1 \otimes c_2) \right\|_{(\mathrm{S}\,n)}.$$

The change is reflected in the Agda proof, which now becomes

```
⊗̂-unit-l : (P : prshf i C) → ∀ x → fst ((Î ⊗̂ P) x) ≃ fst (P x)
⊗̂-unit-l P x =

  fst ((Î ⊗̂ P) x)

    ≃⟨ Trunc-emap (S n) perm ⟩

  Trunc
    (S n)
    (Σ C (λ c₂ → fst (P c₂) × Σ C (λ c₁ → (x == (c₁ ⊗ c₂)) × (c₁ == e))))

    ≃⟨ Trunc-emap (S n)
        (Σ-emap-r (λ c₂ →
         ×-emap-r
          (fst (P a₂))
          ((density (λ c → ((x == (c ⊗ c₂)) , p x (c ⊗ c₂))) e)⁻¹))) ⟩

  Trunc (S n) (Σ C (λ c₂ → fst (P c₂) × (x == (e ⊗ c₂))))

    ≃⟨ Trunc-emap (S n)
        (Σ-emap-r (λ c₂ →
        ×-emap-r
         (fst (P c₂))
         (!-equiv ∘e coe-equiv (⊗-unit-l |in-ctx (λ p → x == p))))) ⟩

  Trunc (S n) (Σ C (λ c₂ → fst (P c₂) × (c₂ == x)))

     ≃⟨ Trunc-emap (S n) (density P x)⁻¹ ⟩

  Trunc (S n) (fst (P x))

     ≃⟨ unTrunc-equiv (fst (P x)) (snd (P x)) ⟩

  fst (P x)

     ≃∎
```

Each step is now performed within a Trunc–emap (S n) which uses an equivalence within the truncation. Then there is an additional last step that removes the truncation, which is allowed because P is a $\mathcal{U}_{(\mathrm{S}\,n)}$-valued presheaf.

4.6 Agda Formalisation

4.6.1 $\mathcal{U}$-presheaves

Concept	Agda Name
$\mathcal{U}$-presheaf	`prshf`
Nat	`nat`

4.6.2 Yoneda

Concept	Agda Name
Yoneda Functor	`yon`
Yoneda Lemma	`yoneda-lemma`
Yoneda Embedding	`yoneda-embedding`
Density Formula	`density`

4.6.3 Cartesian Closure of $\widehat{C}$

Concept	Agda Name
Nat Identity	`id`$_n$
Nat Composition	`_ ∘`$_n$` _`
Presheaf Product	`prod`
Product Equivalence	`prod-equiv`
Presheaf Sum	`sum`
Sum Equivalence	`sum-equiv`
Presheaf Exponential	`_ ⇒ _`
Exponential Equivalence	`⇒-curry`
Terminal Presheaf	`⊤-prshf`
Terminal Universal Morphism	`⊤-prshf-morphism`
Terminal Universal Uniqueness	`⊤-prshf-uniqueness`
Initial Presheaf	`⊥-prshf`
Initial Universal Morphism	`⊥-prshf-morphism`
Initial Universal Uniqueness	`⊥-prshf-uniqueness`

4.6.4 Left Kan Extension

Concept	Agda Name
Left Kan Extension	`lan-y`
Lan_Y of Yoneda	`lan-y-yon`
Lan_Y Composed with Yoneda	`lan-y-∘-yon`
Composition of Lan_Ys	`lan-y-∘`

4.6.5 Day Convolution

Concept	Agda Name
Commutative Monoid	`IsCommMonoid`
Day Tensor	`_ `$\hat{\otimes}$` _`
Day Unit	$\hat{\mathrm{I}}$
Day Unit Left	$\hat{\otimes}$`-unit-l`
Day Unit Right	$\hat{\otimes}$`-unit-r`
Day Associativity	$\hat{\otimes}$`-assoc`
Day Commutativity	$\hat{\otimes}$`-comm`
Day Convolution	`day-convolution`
Yoneda Homomorphism Unit	`yon-hom-unit`
Yoneda Homomorphism Mult	`yon-hom-mult`
Lan_Y Homomorphism Unit	`lan-y-hom-unit`
Lan_Y Homomorphism Mult	`lan-y-hom-mult`

4.6.6 $\mathcal{U}_n$-Presheaves

All the of the concepts of the last five subsections have also been implemented for $\mathcal{U}_n$-presheaves.

5 Free Symmetric Monoid

For a small category $\mathbb{C}$, $\mathbf{SM}\ \mathbb{C}$ is its free strict symmetric-monoidal completion. This can be explicitly defined as the category whose objects are finite sequences $\llparenthesis c_i \rrparenthesis_{i=1,\dots,n}$ of objects of $\mathbb{C}$ and whose homs from $\llparenthesis c_i \rrparenthesis_{i=1,\dots,n}$ to $\llparenthesis c'_j \rrparenthesis_{j=1,\dots,m}$ are only non-empty when $n = m$ and consist of pairs of a bijection $\sigma \in \mathfrak{S}_n$ and a sequence of maps $\llparenthesis f_i : c_i \to c'_{\sigma i} \rrparenthesis_{i=1,\dots,n}$ in $\mathbb{C}$. The translation of this $\mathbf{SM}$ operator into the type theory is a significant focus of this project.

This section discusses our first attempt at formalisation. The idea is to take the type $\mathrm{List}\ \boldsymbol{C}$, in which elements are ordered sequences of elements of $\boldsymbol{C}$ and, to get the morphisms as the bijections between these sequences, take the quotient of this type by the relation of list permutations. Taking the quotient requires the use of Higher Inductive Types (HITs).

5.1 Higher Inductive Types

HITs are one of homotopy type theory's key contributions and most active research areas. Type theory uses the relatively simple framework of inductive types to reason about inductive structures. Although simple, inductive types are powerful reasoning tools, because one can automatically extract induction principles directly from their definitions.

Homotopy theory uses a similar idea of inductively defined spaces. These are defined by not only a collection of points but also a collection of paths or even higher paths. When this idea is taken into homotopy type theory, where paths are now identities, one can extract induction principles for these also. This provides a new way of understanding standard constructions of homotopy theory.

For a given HIT there are two forms of induction principle: the elimination rule, for the dependent case, and the simpler recursion rule, for the non-dependent case. The latter is usually defined in terms of the former. These allow the definition of functions where the domain is a HIT and are necessary because one cannot pattern match on, or deconstruct HITs otherwise.

5.2 Quotients

Given a type C and a relation, $R : C \to C \to \mathcal{U}$, on this type we can form the quotient of C by this relation. This can be defined as the following HIT

$$\begin{aligned}
&\texttt{HIT}\ \mathrm{Quot}_C(R) :\equiv \\
&\quad \mathrm{q} : C \to \mathrm{Quot}_C(R) \\
&\quad \mathrm{rel} : \Pi_{(x,y:C)}\ R\ x\ y \to \mathrm{q}\, x = \mathrm{q}\, y
\end{aligned} \tag{5.1}$$

There is a collection of points $\mathrm{q}\, c$, with one point for each point $c : C$. However, if two points are related according to the relation R, then they are considered equal in the type $\mathrm{Quot}_C(R)$. This is actually a quotient by R^*, the reflexive-symmetric-transitive closure of R. It is reflexive because we always have $\mathrm{idp} : \mathrm{q}\, x = \mathrm{q}\, x$. It is symmetric because if we have $r : R\ x\ y$, but not $R\ y\ x$ in the relation, we still get $!\ (\mathrm{rel}\, r) : \mathrm{q}\, y = \mathrm{q}\, x$. Transitivity similarly comes from path composition.

The elimination principle can be extracted from the type above by forcing the resulting property to reflect the structure of the HIT. Specifically, it has type

$$\begin{aligned}
\mathrm{elim}_{\mathrm{Quot}_C(R)} :\ & \Pi_{(P : \mathrm{Quot}_C(R) \to \mathcal{U})} \\
& \Pi_{(\mathrm{q}^* : \Pi_{(c:C)}\ P\ (\mathrm{q}\, c))} \\
& (\Pi_{(x,y:C)} \Pi_{(r : R\ x\ y)} \to \mathrm{q}^*\, x = \mathrm{q}^*\, y\ [P \downarrow \mathrm{rel}\, x\ y\ r]) \\
& \to \Pi_{(p : \mathrm{Quot}_C(R))}\ P\ p,
\end{aligned}$$

which says that to define a dependent function from $\mathrm{Quot}_C(R)$ to some property $P : \mathrm{Quot}_C(R) \to \mathcal{U}$ we need a point in the resulting property for every point in $\mathrm{Quot}_C(R)$ and a dependent path over every rel path.

This simplifies, in the non-dependent case, to the recursion principle,

$$\begin{aligned}
\mathrm{rec}_{\mathrm{Quot}_C(R)} :\ & \Pi_{(D:\mathcal{U})} \\
& \Pi_{(\mathrm{q}^* : C \to D)} \\
& (\Pi_{(x,y:C)}\ R\ x\ y \to \mathrm{q}^*\, x = \mathrm{q}^*\, y) \\
& \to \mathrm{Quot}_C(R) \to D.
\end{aligned}$$

Set Quotients In Section 5.4 we will define a model of the SM construction for **hSet**s, but to do this the result of applying SM should be an **hSet**. Equation (5.1) says nothing about the higher path structure of the resulting type, so there is no guarantee that it will be a set. Therefore the following HIT is more appropriate

$$\begin{aligned}
&\texttt{HIT}\ \mathrm{SetQuot}_C(R) :\equiv \\
&\quad \mathrm{q} : C \to \mathrm{SetQuot}_C(R) \\
&\quad \mathrm{rel} : \Pi_{(x,y:C)}\ R\ x\ y \to \mathrm{q}\, x = \mathrm{q}\, y \\
&\quad \mathrm{SetQuot\text{-}is\text{-}set} : \mathrm{is\text{-}set}\ \mathrm{SetQuot}_C(R)
\end{aligned} \tag{5.2}$$

The extra constraint of type is-set, defined in Section 2.2, ensures that all path structure above the set level is collapsed and the resulting type can be treated as a set. This change influences the elimination and recursion principles. The first becomes

$$\begin{aligned}\mathrm{elim}_{\mathrm{SetQuot}_C(R)} :\ & \Pi_{(P : \mathrm{SetQuot}_C(R) \to \mathcal{U})} \\ & \Pi_{(\mathrm{q}^* : \Pi_{(c:C)}\ P\ (\mathrm{q}\,c))} \\ & (\Pi_{(x,y:C)} \Pi_{(r : R\ x\ y)} \to \mathrm{q}^*\ x = \mathrm{q}^*\ y\ [P \downarrow \mathrm{rel}\, x\ y\ r]) \\ & \to (\Pi_{(c:C)}\ \text{is-set}\ P\ c) \\ & \to \Pi_{(p : \mathrm{SetQuot}_C(R))}\ P\ p\end{aligned}$$

adding a proof of is-set for every resulting property and the second becomes

$$\begin{aligned}\mathrm{rec}_{\mathrm{SetQuot}_C(R)} :\ & \Pi_{(D:\mathcal{U})} \\ & \Pi_{(\mathrm{q}^* : C \to D)} \\ & (\Pi_{(x,y:C)}\ R\ x\ y \to \mathrm{q}^*\ x = \mathrm{q}^*\ y) \\ & \to \text{is-set}\ D \\ & \to \mathrm{SetQuot}_C(R) \to D\end{aligned}$$

adding a proof of is-set D.

Groupoid Quotients In section 5.5 we will extend the SM model to further work for **hGpd**s, which again requires a change to the quotient. The type will be restricted to being a groupoid rather than a set, which means there will be more path structure. Modifying eq. (5.3) below to use is-gpd, defined in Section 2.2, gives one definition of a quotient at the groupoid level, but it is not general enough for our purposes. This definition would again be quotienting by R^*, but this time, because we are working with a groupoid, the reflexivity and transitivity paths do not necessarily coincide with the ones that R may already have. To generalise, given an R along with

$$\text{R-is-refl} : \Pi_{(c:C)}\ R\ c\ c,$$

a proof of reflexivity, and

$$\text{R-is-trans} : \Pi_{(x,y,z:C)}\ R\ x\ y \to R\ y\ z \to R\ x\ z,$$

a proof of transitivity, we want to force the former to coincide with idp and the latter to coincide with $\cdot$. This leads to the following HIT

$$\begin{aligned}&\mathtt{HIT}\ \mathrm{GpdQuot}_C(R) :\equiv \\ &\quad \mathrm{q} : C \to \mathrm{GpdQuot}_C(R) \\ &\quad \mathrm{rel} : \Pi_{(x,y:C)}\ R\ x\ y \to \mathrm{q}\, x = \mathrm{q}\, y \\ &\quad \text{rel-idp} : \Pi_{(c:C)}\ \mathrm{rel}\, c\ c\ (\text{R-is-refl}\, c) = \mathrm{idp} \\ &\quad \text{rel-}\cdot : \Pi_{(x,y,z:C)} \Pi_{(r:R\ x\ y)} \Pi_{(r':R\ y\ z)}\ \mathrm{rel}\, x\ y\ r \cdot \mathrm{rel}\, y\ z\ r' = \mathrm{rel}\, x\ z\ (\text{R-is-trans}\, r\ r') \\ &\quad \text{GpdQuot-is-gpd} : \text{is-gpd}\ \mathrm{GpdQuot}_C(R)\end{aligned} \tag{5.3}$$

We not that one does not need to add symmetry, which would be given by

$$\Pi_{(x,y:C)}\Pi_{(r:R\,x\,y)}\,!(\mathrm{rel}\,x\,y\,r) = \mathrm{rel}\,y\,x\,(\text{R-is-sym}\,r)$$

because it can be derived from the other two. We have

$$\begin{aligned}\mathrm{rel}\,x\,y\,r\cdot!(\mathrm{rel}\,x\,y\,r) &= \mathrm{idp}\\ &= \mathrm{rel}\,x\,x\,(\text{R-is-refl}\,x)\\ &= \mathrm{rel}\,x\,x\,(\text{R-is-trans}\,r\,(\text{R-is-sym}\,r))\\ &= \mathrm{rel}\,x\,y\,r\cdot\mathrm{rel}\,y\,x\,(\text{R-is-sym}\,r)\end{aligned}$$

and then

$$\begin{aligned}!(\mathrm{rel}\,x\,y\,r) &=!(\mathrm{rel}\,x\,y\,r)\cdot\mathrm{rel}\,x\,y\,r\cdot!(\mathrm{rel}\,x\,y\,r)\\ &=!(\mathrm{rel}\,x\,y\,r)\cdot\mathrm{rel}\,x\,y\,r\cdot\mathrm{rel}\,y\,x\,(\text{R-is-sym}\,r)\\ &= \mathrm{rel}\,y\,x\,(\text{R-is-sym}\,r).\end{aligned}$$

The elimination and recursion principles need to reflect these new paths. The elimination principle becomes

$$\begin{aligned}\mathrm{elim}_{\mathrm{GpdQuot}_C(R)} :\ &\Pi_{(P:\mathrm{GpdQuot}_C(R)\to\mathcal{U})}\\ &\Pi_{(\mathrm{q}^*:\Pi_{(c:C)}\,P\,(\mathrm{q}\,c))}\\ &\Pi_{(\mathrm{rel}^*:\Pi_{(x,y:C)}\Pi_{(r:R\,x\,y)}\to\mathrm{q}^*\,x=\mathrm{q}^*\,y\,[P\downarrow\mathrm{rel}\,x\,y\,r])}\\ &(\Pi_{(c:C)}\,\mathrm{rel}^*\,c\,c\,(\text{R-is-refl}\,c) = \mathrm{idp}^d\ [(\lambda p\to\mathrm{q}^*\,c=\mathrm{q}^*\,c\,[P\downarrow p])\downarrow\text{rel-idp}\,c])\\ &\to(\Pi_{(x,y,z:C)}\Pi_{(r:R\,x\,y)}\Pi_{(r':R\,y\,z)}\\ &\quad\to\mathrm{rel}^*\,x\,y\,r\cdot\mathrm{rel}^*\,y\,z\,r' = \mathrm{rel}^*\,x\,z\,(\text{R-is-trans}\,r\,r')\\ &\qquad[(\lambda p\to\mathrm{q}^*\,x=\mathrm{q}^*\,z\,[P\downarrow p])\downarrow\text{rel-}\cdot\,x\,y\,z\,r\,r']\\ &\to(\Pi_{(c:C)}\ \text{is-gpd}\,P\,c))\\ &\to\Pi_{(p:\mathrm{GpdQuot}_C(R))}\,P\,p\end{aligned}$$

This is quite complex, but can be derived mechanically from the definition of the HIT. Compared to the elimination principle for $\mathrm{SetQuot}_C(R)$ there are two new families of dependent paths, lying over rel-idp and rel-· respectively. Again this simplifies to the recursion principle

$$\begin{aligned}\mathrm{rec}_{\mathrm{GpdQuot}_C(R)} :\ &\Pi_{(D:\mathcal{U})}\\ &\Pi_{(\mathrm{q}^*:C\to D)}\\ &\Pi_{(\mathrm{rel}^*:\Pi_{(x,y:C)}\,R\,x\,y\to\mathrm{q}^*\,x=\mathrm{q}^*\,y)}\\ &(\Pi_{(c:C)}\,\mathrm{rel}^*\,c\,c\,(\text{R-is-refl}\,c) = \mathrm{idp})\\ &\to(\Pi_{(x,y,z:C)}\Pi_{(r:R\,x\,y)}\Pi_{(r':R\,y\,z)}\\ &\quad\to\mathrm{rel}^*\,x\,y\,r\cdot\mathrm{rel}^*\,y\,z\,r' = \mathrm{rel}^*\,x\,z\,(\text{R-is-trans}\,r\,r'))\\ &\to\text{is-gpd}\,D\\ &\to\mathrm{GpdQuot}_C(R)\to D\end{aligned}$$

which says that to define a function $\mathrm{GpdQuot}_C(R) \to D$ we need to show that D is a groupoid and further that q^* preserves the structure of the relation, i.e. relates the appropriate elements and has reflexivity and transitivity.

The computation of these HITs is defined in Agda using a rewrite rule. For quotients the rule says

$$\mathrm{elim}_{\mathrm{GpdQuot}_C(R)} \dots (\mathrm{q}\, c) \mapsto \mathrm{q}^*\, c$$

where … represents the arguments to the elimination rule.

5.3 List Permutations

To define list permutations we will need a couple of auxilliary definitions. The first is the predicate of list membership $\in$ which for elements $c, c' : C$ and a list $l : \mathrm{List}\, C$ has constructors

$$\mathrm{here} : c == c' \to c' \in (c :: l)$$

and

$$\mathrm{there} : c' \in l \to c' \in (c :: l).$$

These act as an index into the list $c :: l$ giving a position for the element c' as a sequence of theres and a final here. The second is the remove function

$$\mathrm{remove} : \Pi_{(x:C)} \Pi_{(xs:\mathrm{List}\, C)}\ x \in xs \to \mathrm{List}\, C$$

which traverses the list following the sequence of theres and then removes the element that corresponds to the here.

Using these, we define list permutations to be the inductive relation

$$\begin{aligned}
&\textbf{data}\ \mathrm{ListPerm} : \mathrm{List}\, C \to \mathrm{List}\, C \to \mathcal{U}\ \textbf{where}\\
&\quad \mathrm{lpnil} : \mathrm{ListPerm}\, \mathrm{nil}\, \mathrm{nil}\\
&\quad \mathrm{lpcons} : \Pi_{\{x:C\}} \Pi_{\{xs,xs':\mathrm{List}\, C\}} \Pi_{(e:x\in xs')}\ \mathrm{ListPerm}\, xs\, (\mathrm{remove}\, xs'\, e) \to \mathrm{ListPerm}\, (x :: xs)\, xs'
\end{aligned}$$

In the base case the empty list is a permutation of itself. In the inductive case, the list $x :: xs$ is a permutation of the list xs' if $x \in xs'$ and xs is a permutation of xs' with x removed. This ensures that each element of one list appears exactly once in the other.

List permutations form an equivalence relation between lists on a type. Therefore they satisfy reflexivity,

$$\mathrm{ListPerm\text{-}is\text{-}refl} : \Pi_{(c:\mathrm{List}\, C)}\ \mathrm{ListPerm}\, c\, c,$$

symmetry,

$$\mathrm{ListPerm\text{-}is\text{-}sym} : \Pi_{(x,y:\mathrm{List}\, C)}\ \mathrm{ListPerm}\, x\, y \to \mathrm{ListPerm}\, y\, x,$$

and transitivity,

$$\text{ListPerm-is-trans} : \Pi_{(x,y,z:\text{List}\,C)}\ \text{ListPerm}\,x\ y \to \text{ListPerm}\,y\ z \to \text{ListPerm}\,x\ z.$$

Reflexivity is simple. For an empty list we have lpnil. For a list $x :: xs$ we apply lpcons to here idp and an inductive call to reflexivity. This is the identity permutation on a list.

Symmetry, corresponding to inverse permutations, is more involved. Consider the cases for an input σ. For $\sigma \equiv \text{lpnil}$ we of course have lpnil. For $\sigma \equiv \text{lpcons}\,(\text{here}\,\text{idp})\,\sigma'$ we have $\text{lpcons}\,(\text{here}\,\text{idp})\,(\text{ListPerm-is-sym}\,\sigma')$. In the case of $\sigma \equiv \text{lpcons}\,(\text{there}\,e)\,\sigma'$ we have to do more. The permutation will look something like

$$\begin{tikzcd}
x_1 \arrow[drrr] & x_2 \arrow[d] & x_3 \arrow[dll] & x_4 \arrow[drr] & \cdots & x_n \arrow[dlll] \\
y_1 & y_2 & y_3 & y_4 & \cdots & y_n
\end{tikzcd}$$

and we want to define the inverse permutation, meaning we need to give a mapping for y_1. Notice that if we remove the pair (x_1, y_4), then the mapping for x_2 is a there. Then, removing the pair (x_2, y_2), the mapping for a_3 is a here. In general, to find the index for y_1 we count the distance to the first here value in the permutation, using the function first-here. Then the rest of the permutation should be a recursive call, applying symmetry to the permutation with the pair (x_3, y_1) removed. To remove this pair the indices of the first two pairs should be reduced. This is done using the remove-first-here function. The whole definition becomes

$$\text{lpcons}\,(\text{first-here}\,\sigma)\,(\text{ListPerm-is-sym}\,(\text{remove-first-here}\,\sigma)).$$

Finally transitivity, which corresponds to composition of permutations and which we'll write $\sigma_2 \circ \sigma_1$. Intuitively we just need to join up the lines of diagrams like the ones above. However, formalising this concept was one of the significant challenges of this project. So we have $\sigma_1 : \text{ListPerm}\,x\,y$ and $\sigma_2 : \text{ListPerm}\,y\,z$. In the base case we have both permutations being lpnil and the result is of course lpnil. The next simplest case is the one where $\sigma_1 \equiv \text{lpcons}\,(\text{here}\,\text{idp})\,\sigma_1'$ and $\sigma_2 \equiv \text{lpcons}\,e\,\sigma_2'$ because here the resulting permutation is simply $\text{lpcons}\,e\,(\sigma_2' \circ \sigma_1')$.

The remaining two cases have the first permutation as $\sigma_1 \equiv \text{lpcons}\,(\text{there}\,e)\,\sigma_1'$. In both cases the first argument to lpcons is the index got following the two indices. To calculate this value we have the function

$$\in\text{-ListPerm} : x \in xs \to \text{ListPerm}\,xs\,ys \to x \in ys$$

such that $\in\text{-ListPerm}\,(\text{there}\,e)\,\sigma_2$ gives us the index of the first element of x in z. We want to call $\circ$ recursively and the second argument will be σ_1. We need to consider the two cases for the first argument separately.

For the first case we have $\sigma_2 \equiv \text{lpcons}\,(\text{here idp})\,\sigma_2'$. σ_1 had a there at the head, so we will not be removing the first value of the second list. Therefore, the first argument to $\circ$ starts with $\text{lpcons}\,(\text{here idp})$. Now the tail of this list has to remove the pair which was used to join the first element. This removal is the most complicated part of the algorithm, so we will not go into the details.

For the second case we have $\sigma_2 \equiv \text{lpcons}\,(\text{there}\,e)\,\sigma_2'$. We cannot guarantee that the result starts with the same there e this time, so we instead call the removal function on the whole lists. Again the details of this are omitted.

5.4 hSet Model

The definition of the free strict symmetric-monoidal completion given at the beginning of this chapter was for categories. Before trying to model this for our interpretation of types as groupoids, we will first look at modelling the set-theoretic version of the same construction. This corresponds to the finite-multiset construction on sets.

From the definition at the beginning of the chapter, the construction on categories had objects as finite sequences of objects of the underlying category, and homs as bijections between these sequences. We can view the category as a set by collapsing the homs to having at most one morphism. Now there will be morphisms between two objects if and only if there is a bijection between them. There will now be fully connected components, each of which corresponds to a multiset.

Working with multisets, we get the notion of the empty multiset, $\varnothing$, corresponding to an empty sequence, and the union of multisets, $\cup$, corresponding to concatenation of sequences. These are the unit and associative operation of a monoid. We also get the singleton multiset containing c, $[c]$, corresponding to a singleton sequence.

To be the *free* construction the definition must have the corresponding universal property, which says that for every set C and commutative monoid M, a function $f : C \to M$ uniquely extends to a function $f^{\#} : \text{SM}\ C \to M$ that is a monoid homomorphism, i.e. it satisfies the unit law

$$f^{\#}\ \varnothing = e$$

and the multiplication law

$$f^{\#}\ (x \cup y) = f^{\#}\ x \otimes f^{\#}\ y,$$

where e is the unit of M and $\otimes$ its commutative, associative operation, and further satisifies the singleton law

$$f^{\#}\ [c] = f\ c.$$

It should also satisfy a uniqueness property that for any other monoid homomorphism h such that

$$h\ [c] = f\ c$$

we should have that

$$h = f^{\#}.$$

The **hSet** model of this finite multiset construction is obtained as

$$\mathrm{SM}\, C = \mathrm{SetQuot}_{\mathrm{List}\, C}(\mathrm{ListPerm}\, C),$$

quotienting the type of lists on C by the relation of list permutations. This formalisation makes it easy to define the empty multiset

$$\varnothing :\equiv \mathrm{q}\,\mathrm{nil}$$

and the singleton

$$[c] :\equiv \mathrm{q}\ (c\ ::\ \mathrm{nil}).$$

The next step is to define the union operation, which combines two values of SM C. Defining a function where the domain is a HIT requires the use of the associated induction principle. The union operation is non-dependent, so the definition uses the recursion rule for quotients defined above.

```
_∪_ : SM C → SM C → SM C
_∪_ =
  SM-rec
    (→-is-set SM-is-set)
    (λ x →
      SM-rec
        SM-is-set
        (λ y → qp[ x ++ y ])
        (λ r → quot-rel (ListPerm-cong-++-r x r)))
    (λ r →
      λ=
        (SM-elim
          (λ _ → =-preserves-set SM-is-set)
          (λ y → quot-rel (ListPerm-cong-++-l r y))
          (λ _ → prop-has-all-paths-↓ (SM-is-set _ _))))
```

Here the q from the quotient has become the operator `qp[_]` and the rel has become `quot-rel`, because the set quotient was already defined in HoTT-Agda with these names.

The union operation is a function of two arguments, so the recursion principle needs to be used twice. For the first application, the result is a function type. Function types have the same level as their codomain, so the whole function is a set because SM produces a set. To define q* for this operation we use the recursion rule a second time, with the first argument already in scope. The result is SM C, which is a set. The q* at this second level defines the actual functionality of the union operation, which is to append the two underlying lists and then apply the quotient.

The rest of the definition is the proofs rel^* for each level. The inner rel^* is a congruence condition that says

$$\Pi_{(x,y,z:\mathrm{List}\ C)}\Pi_{(r:\mathrm{ListPerm}\ \ y\ z)}\ \mathrm{ListPerm}\ (x ++ y)\ (x ++ z).$$

The outer rel^* requires an application of the elimination rule because it involves a dependent path. This itself requires another rel^* proof, but this one is at the level above set, so follows trivially because SM is a set. This trivialisation of higher rel^* proofs was part of the motivation for starting with the set theoretic model.

Given a function $f : C \to M$ for a commutative monoid M, the universal property uniquely extends to a function $f^{\#} : \mathrm{SM}\,C \to M$. Functions from $\mathrm{SM}\,C$ are defined using the recursion principle by giving a function on the underlying list, $\mathrm{q}^* : \mathrm{List}\ C \to D$. For this we define the function $\text{monoid-fold} : (C \to D) \to \mathrm{List}\ C \to D$ as

$$\text{monoid-fold}\ f :\equiv \mathrm{foldr}\ (\lambda\ x\ m \to f\ x \otimes m)\ e$$

which applies f to each element of the list and joins the results using M's associative operation. This must come with an associated $\mathrm{rel}^* : \Pi_{(x,y:C)}\ \mathrm{ListPerm}\,x\ y \to \mathrm{q}^*\ x = \mathrm{q}^*\ y$, which we get using the function

$$\text{monoid-fold-is-const} : \Pi_{(f:C\to M)}\Pi_{(x,y:C)}\ \mathrm{ListPerm}\,x\ y \to \text{monoid-fold}\,f\ x = \text{monoid-fold}\,f\ y.$$

This leads to the following definition of the universal property

$$f^{\#} :\equiv \mathrm{rec}_{\mathrm{SM}}\ M\ (\text{monoid-fold}\,f)\ (\text{monoid-fold-is-const}\,f).$$

Finally we show that this definition satisfies the necessary conditions. The unit law follows from the definitions of $\varnothing$ and monoid-fold, as

$$\text{monoid-fold}\,\mathrm{nil} :\equiv e.$$

The singleton law again follows computationally, as

$$\text{monoid-fold}\ (c :: \mathrm{nil}) :\equiv f\ c.$$

To prove the multiplication law we use the elimination rule twice, with the inner q^* being the function

$$\text{monoid-fold-is-}{++} : \Pi_{(x,y:\mathrm{List}\ C)}\,\text{monoid-fold}\ (x ++ y) = \text{monoid-fold}\,x \otimes \text{monoid-fold}\,y,$$

which performs induction on x. When x is nil, then this is simply the left unit law for the monoid. When x is $x :: xs$ this reduces to showing

$$f\ x \otimes \text{monoid-fold}\ (xs ++ y) = (f\ x \otimes \text{monoid-fold}\,xs) \otimes \text{monoid-fold}\,y,$$

which requires a recursive call and an application of associativity of the monoid operation.

To prove uniqueness we use function extensionality and the elimination principle leaving us to show

$$\text{monoid-fold}\, f\ x = h\ (\text{q}\, x)$$

Again we use induction on x. In the nil case we can use the unit law for h. In the $x :: xs$ case we have

$$\begin{aligned}
\text{monoid-fold}\ (x :: xs) &= f\ x \otimes h\ (\text{q}\, xs) && \text{(induction)}\\
&= h\ [x] \otimes h\ (\text{q}\, xs) && \text{(}h\text{ singleton law)}\\
&= h\ \text{q}\ (x :: xs) && \text{(}h\text{ multiplication law)}
\end{aligned}$$

So the definition does give the desired universal property. Using the universal property the SM construction can be seen as an endofunctor on **hSet**. The action of the SM construction on morphisms can be defined in terms of the universal property. For a function $f : C \to D$,

$$\text{SM}\ f :\equiv ([\_] \circ f)^{\#}.$$

5.5 hGpd Model

In the **hSet** model, for any two values $x, y : \text{SM}\, C$ the path space $x = y$ is a proposition, so all paths are considered equal. This means that, considering $\text{SM}\, C$ as a groupoid every hom contains either no morphisms or one morphism. This restriction removes non-trivial groupoid structure. It is worthwhile to extend the definition to **hGpd**s using $\text{GpdQuot}_C(R)$ from Section 5.2.

Again this HIT allows us to construct the type $\text{SM}\, C$ by applying it to the relation ListPerm over C, extended with reflexivity and transitivity,

$$\text{SM}\, C :\equiv \text{GpdQuot}_{\text{List}\, C}(\text{ListPerm}).$$

This time, however, proofs of simple properties become highly involved. For example, in the definition of the union operation in the **hSet** model, we only had to define rel* for the inner application of the recursion principle, getting the outer one for free. Now we have to further define rel-idp* and rel-·* for the inner application, and rel* for the outer application, getting only the last two paths for the outer application trivially.

5.6 Agda Formalisation

5.6.1 Quotients

Concept	Agda Name
Groupoid Quotient Elimination Principle	`GpdQuot-elim`
Groupoid Quotient Recursion Principle	`GpdQuot-rec`

5.6.2 List Permutations

Concept	Agda Name
remove	`remove`
ListPerm	`ListPerm`
ListPerm Reflexivity	`id-perm`
first-here	`first-here`
remove-first-here	`remove-first-here`
ListPerm Symmetry	`inv-perm`
∈-ListPerm	`∈-ListPerm`
ListPerm Transitivity	$\_ \circ_p \_$
ListPerm Concatenation	$\_ +\!\!+_p \_$
ListPerm ++ Congruence Left	`ListPerm-cong-++-l`
ListPerm ++ Congruence Right	`ListPerm-cong-++-r`

5.6.3 hSet Model

Concept	Agda Name
SM	`SM`
∅	`∅`
[_]	`SM-⊤`
∪	`_∪_`
Union Unit Left	`∪-unit-l`
Union Unit Right	`∪-unit-r`
Union Associativity	`∪-assoc`
Union Commutativity	`∪-comm`
SM Monoid	`SM-is-comm-monoid`
monoid-fold	`monoid-fold`
monoid-fold-is-const	`monoid-fold-is-const`
Universal Extension	`up`
Universal Homomorphism Unit	`up-hom-unit`
Universal Homomorphism Mult	`up-hom-mult`
Universal Applied to Singleton	`up-hom-⊤`
Universal Uniqueness	`up-uniqueness`
SM Functor on Morphisms	`SM-fmap`

5.6.4 hGpd Model

The **hGpd** model defines all the same concepts as the **hSet** model. However, due to the difficulty of coherence proofs at the groupoid level the following definitions are or rely on postulated results.

6 A Truncated Formalisation

6.1 Imposed Finiteness

The first formalisation of the free symmetric-monoidal completion was based on lists over some type C. These lists are finite by construction. The desired morphisms, i.e. permutations between these lists, were added directly using the quotient HIT. Alternatively lists over C can be described using some finite indexing type I and a function $f : I \to C$, leading to the definition

$$\mathrm{SM}_0\, C :\equiv \Sigma_{(I : \mathcal{U})}\, (I \to C) \times \Sigma_{(n : \mathbb{N})}\, I \simeq \mathrm{Fin}\, n.$$

Unfortunately this definition doesn't quite work, because the path space does not have the correct structure. For this type, the path space contains not only equality of the underlying indexing types and functions, but also of the proof of finiteness. This forces the types to be finite in the same way, trivially restricting the path space. To get around this problem the proofs of finiteness need to be ignored. This can be achieved using propositional truncation, the truncation operation at level -1. Truncating the proof of finiteness means that two different proofs are always equal, so no longer affect the path space. The final definition becomes

$$\mathrm{SM}\, C :\equiv \Sigma_{(I : \mathcal{U})}\, (I \to C) \times \Sigma_{(n : \mathbb{N})}\, \| I \simeq \mathrm{Fin}\, n \|.$$

This is based on the definition of finite universes in Yorgey [12] and generalises it for our purposes.

Again we can define the empty value and the singleton value, although they are no longer as simple as the previous formalisation. The empty value is

$$\varnothing :\equiv (\bot, \bot\text{-rec}, 0, |\, \mathrm{Fin0}\text{-}\bot\, |)$$

where $\bot$ is the empty type, $\bot$-rec is the unique function from the empty type to C, and Fin0-$\bot$ is a proof that $\mathrm{Fin}\, 0$ is equivalent to the empty type. The singleton value is

$$[c] :\equiv (\top, \lambda\, \_ \to c, 1, |\, \mathrm{Fin1}\text{-}\top\, |)$$

where $\top$ is the unit type, and Fin1-$\top$ is a proof that $\mathrm{Fin}\, 1$ is equivalent to the unit type. These clearly have the desired behaviour, because the empty value picks out no C values and $[c]$ picks out the specific value c.

The union operation has type $\mathrm{SM}\, C \to \mathrm{SM}\, C \to \mathrm{SM}\, C$, so needs to combine two values of this Σ type. Take two values of the type: (I, i, m, ti) and (J, j, n, tj). To combine the first components is to create a new indexing set containing all the values of the two initial ones. This can be done using the coproduct which corresponds to a disjoint union of the types. The indexing function now has to perform case analysis on a value of type $I \sqcup J$. When the value is in I we want to apply i and when it is in J, j. This can be written $[i, j]$. $I \sqcup J$ clearly has size $m + n$, so finally we have to combine the two proofs of finiteness. To do this we first have to show

$$\sqcup\text{-finite} : I \simeq \mathrm{Fin}\, m \to J \simeq \mathrm{Fin}\, n \to I \sqcup J \simeq \mathrm{Fin}\,(m + n).$$

This can then be turned into a function on truncated values using Trunc-fmap2, which has type

$$(C \to D \to E) \to \|C\| \to \|D\| \to \|E\|,$$

and uses the truncation recursion principle twice, then simply applies the supplied function inside a truncation. So the final definition is

$$(I, i, m, ti) \cup (J, j, n, tj) :\equiv (I \sqcup J, [i, j], m + n, \text{Trunc-fmap2}\, \sqcup\text{-finite}\; ti\; tj).$$

Showing that $\mathrm{SM}\, C$ forms a commutative monoid with unit $\varnothing$ and associative operation $\cup$ reduces to showing the commutative monoid laws for the monoid with unit $\bot$ and associative operation $\sqcup$ and for the monoid with unit 1 and associative operation $+$.

We want to define the universal property for this truncated definition, but to do so we need to define a function $\mathrm{SM}\, C \to M$, where M is a monoidal groupoid. The type $\mathrm{SM}\, C$ contains a truncated value and truncation is defined as a HIT. Therefore, to use this value we need to use an appropriate recursion principle. The next section discusses such a principle.

6.2 Functions from Propositional Truncations

The usual recursion principle for propositional truncation has the type,

$$(C \to D) \to \text{is-prop}\, D \to \|C\| \to D.$$

So to construct a function out of a propositional truncation, the result has to be a proposition. Any definition of the universal property for the truncated $\mathrm{SM}\, C$ will have to use the truncated equivalence stored inside it, but we would like the result to be a groupoid, not a proposition. Therefore this recursion principle is not strong enough to allow us to define the universal property in its full generality.

The intuition behind using truncated values is that the result should not leak information hidden by the truncation. For example if C is a set and has the higher path $p =_{c =_C c'} p'$, any function that uses a value of $\|C\|$ should not allow the knowledge that this path exists to leak into the result type. One way to enforce this is to force the result type to have the same truncation level, as is done in the above recursion principle. However, there should be some functions that do not leak information and yet have higher level results, the obvious example being a constant function to a type of higher level.

Intuitively, not leaking information corresponds to not caring what the value is, only that it exists, so we might want for some function $f : C \to D$,

$$\Pi_{(c,c':C)}\, f\ c = f\ c'. \tag{6.1}$$

We conjected that we should be able to use this property and found that Kraus [8] had indeed proved the equivalence

$$(\|C\| \to D) \simeq \Sigma_{(f:C\to D)}\ \text{is-wconst}\ f,$$

for D a set, where is-wconst is exactly the property above, called weak-constancy. This property is not enough for the groupoid case however. The following path is needed to force higher paths to behave correctly in the groupoid case,

$$\text{is-coh}\ f\ c :\equiv \Pi_{(x,y,z:C)}\ c\ x\ y \cdot c\ y\ z = c\ x\ y \tag{6.2}$$

and so the equivalence becomes

$$(\|C\| \to D) \simeq \Sigma_{(f:C\to D)}\Sigma_{(c:\text{is-wconst}\ f)}\ \text{is-coh}\ f\ c, \tag{6.3}$$

for D a groupoid.

6.3 The Universal Property

The universal extension has type

$$(C \to M) \to \text{SM}\, C \to M$$

for a commutative monoid M. Given some $f : C \to M$ and some $(I, i, m, ti) : \text{SM}\, C$ where I is the indexing type and $i : I \to C$ the indexing function, we need to construct an M. The idea is to apply f to each element indexed by I and then to join these together using M's associative operation. To actually run over the values in I we need to use the backwards direction of the equivalence defined by ti, so eq. (6.3) is necessary.

A function is needed with the following type

$$\text{monoid-fold} : (C \to M) \to (I \to C) \to (I \simeq \text{Fin}\, m) \to M$$

which when applied to f and i gives a function that is weakly-constant and coherent as defined in eqs. (6.1) and (6.2) respectively. Then finally applying the backwards direction of the equivalence to this we get a function

$$\|I \simeq \text{Fin}\, m\| \to M$$

which when applied to ti gives us the required M. The function is defined as

$$\text{monoid-fold}\ f\ i\ eq = \text{monoid-fold}'\ m\ f\ (i \circ \leftarrowtail\ eq)$$

where $\leftarrowtail$ extracts the backwards direction of eq and $\text{monoid-fold}'$ is defined as

$$\begin{aligned}\text{monoid-fold}'\ 0\ \_\ \_\ &:\equiv e\\ \text{monoid-fold}'(\mathrm{S}\ n)\ f\ g\ &:\equiv f\ (g\ \mathtt{fresh}) \otimes \text{monoid-fold}'\ n\ f\ (g \circ \mathrm{inc})\end{aligned}$$

where $\mathtt{fresh}$ is the largest value of a $\mathrm{Fin}\ n$ and inc is the inclusion function of type $\mathrm{Fin}\ n \to \mathrm{Fin}\ (\mathrm{S}\ n)$. So this runs through every value of the Fin.

Constancy Intuitively it seems that this definition should be weakly-constant because $\otimes$ is commutative. For any proof of finiteness all of the values of I will be included in the final value and so the order doesn't really matter. To formally prove this is quite involved.

To prove this we will reduce the problem to showing

$$\begin{aligned}\text{monoid-fold}'\text{-is-perm-invariant} : &\Pi_{(m:\mathbb{N})}\Pi_{(f:I\to C)}\Pi_{(g:\mathrm{Fin}\ n\to I)}\Pi_{(\sigma:\mathrm{Fin}\ n\simeq \mathrm{Fin}\ n)}\\ &\text{monoid-fold}'\ n\ f\ g = \text{monoid-fold}'\ n\ f\ (g \circ \rightarrowtail\ \sigma)\end{aligned}$$

which can then be applied for $eq, eq' : I \simeq \mathrm{Fin}\ n$ as

$$\text{monoid-fold}'\text{-is-perm-invariant}\ n\ f\ (g \circ \leftarrowtail\ eq)\ (eq \circ eq'^{-1}).$$

This gives the path

$$\text{monoid-fold}'\ n\ f\ (g \circ \leftarrowtail\ eq) = \text{monoid-fold}'\ n\ f\ (g \circ \leftarrowtail\ eq \circ \rightarrowtail\ eq \circ \leftarrowtail\ eq')$$

and we can cancel the eq on the right-hand side to get

$$\text{monoid-fold}'\ n\ f\ (g \circ \leftarrowtail\ eq) = \text{monoid-fold}'\ n\ f\ (g \circ \leftarrowtail\ eq')$$

as we want.

To prove $\text{monoid-fold}'\text{-is-perm-invariant}$ we use strong induction. We will join the f and g of $\text{monoid-fold}'$ into a single h without loss of generality. The base cases

$$\text{monoid-fold}'\ 0\ h = \text{monoid-fold}'\ 0\ (h \circ \rightarrowtail\ \sigma)$$

and

$$\text{monoid-fold}'\ (\mathrm{S}\ 0)\ h = \text{monoid-fold}'\ (\mathrm{S}\ 0)\ (h \circ \rightarrowtail\ \sigma)$$

both hold because the only σ is the identity permutation. For the inductive case we have two cases.

First the case that

$$\rightarrowtail \sigma\ \mathrm{fresh} \equiv \mathrm{fresh},$$

i.e. the first element maps to itself. In this case we can immediately make a recursive call, giving us

$$\text{monoid-fold}'\,(\mathrm{S}\,n)\,(h \circ \mathrm{inc}) = \text{monoid-fold}'\,(\mathrm{S}\,n)\,(h \circ \mathrm{inc} \circ \rightarrowtail (\downarrow \sigma)),$$

where $\downarrow : \Pi_{(\sigma : \mathrm{Fin}\,(\mathrm{S}\,n) \simeq \mathrm{Fin}\,(\mathrm{S}\,n))}(\rightarrowtail \sigma\,\mathrm{fresh} = \mathrm{fresh}) \to (\mathrm{Fin}\,n \simeq \mathrm{Fin}\,n)$. We then have the lemma

$$\mathrm{inc} \circ \rightarrowtail (\downarrow \sigma) = \rightarrowtail \sigma \circ \mathrm{inc},$$

giving us

$$\text{monoid-fold}'\,(\mathrm{S}\,n)\,(h \circ \mathrm{inc}) = \text{monoid-fold}'\,(\mathrm{S}\,n)\,(h \circ \rightarrowtail \sigma \circ \mathrm{inc}).$$

This now gives us the result by putting the appropriate elements, which we assumed were equal, on the front of each side.

For the second case we define the transposition permutation

$$\text{trans-perm} : \Pi_{(a,b : \mathrm{Fin}\,n)}\ \mathrm{Fin}\,n \simeq \mathrm{Fin}\,n,$$

where $\text{trans-perm}\,a\,b$ switches a and b, so has definition

$$\text{trans-perm}\,a\,b\,c :\equiv \begin{cases} b, & \text{if } c = a \\ a, & \text{if } c = b \\ c, & \text{otherwise} \end{cases}$$

The actual definition of this permutation requires complicated use of Agda's with-abstraction to compare natural numbers.

Using trans-perm we have the following long chain of identities

$$
\begin{aligned}
&\ \text{monoid-fold}'\,(\mathrm{S}\,(\mathrm{S}\,n))\,h\\
&\equiv h\,(\mathrm{S}\,n) \otimes \text{monoid-fold}'\,(\mathrm{S}\,n)\,(h \circ \mathrm{inc})\\
&\qquad \text{(induction)}\\
&= h\,(\mathrm{S}\,n) \otimes \text{monoid-fold}'\,(\mathrm{S}\,n)\,(h \circ \mathrm{inc} \circ\ \rightarrowtail \text{trans-perm}\,n\,(\rightarrowtail \sigma\,(\mathrm{S}\,n)))\\
&\equiv h\,(\mathrm{S}\,n) \otimes (h\,(\rightarrowtail \sigma\,(\mathrm{S}\,n)) \otimes \text{monoid-fold}'\,n\,(h \circ \mathrm{inc} \circ\ \rightarrowtail \text{trans-perm}\,n\,(\rightarrowtail \sigma\,(\mathrm{S}\,n)) \circ \mathrm{inc}))\\
&\qquad \text{(associativity)}\\
&= (h\,(\mathrm{S}\,n) \otimes h\,(\rightarrowtail \sigma\,(\mathrm{S}\,n))) \otimes \text{monoid-fold}'\,n\,(h \circ \mathrm{inc} \circ\ \rightarrowtail \text{trans-perm}\,n\,(\rightarrowtail \sigma\,(\mathrm{S}\,n)) \circ \mathrm{inc})\\
&\qquad \text{(commutativity)}\\
&= (h\,(\rightarrowtail \sigma\,(\mathrm{S}\,n)) \otimes h\,(\mathrm{S}\,n)) \otimes \text{monoid-fold}'\,n\,(h \circ \mathrm{inc} \circ\ \rightarrowtail \text{trans-perm}\,n\,(\rightarrowtail \sigma\,(\mathrm{S}\,n)) \circ \mathrm{inc})\\
&\qquad \text{(associativity)}\\
&= h\,(\rightarrowtail \sigma\,(\mathrm{S}\,n)) \otimes (h\,(\mathrm{S}\,n) \otimes \text{monoid-fold}'\,n\,(h \circ \mathrm{inc} \circ\ \rightarrowtail \text{trans-perm}\,n\,(\rightarrowtail \sigma\,(\mathrm{S}\,n)) \circ \mathrm{inc}))\\
&\qquad \text{(induction)}\\
&= h\,(\rightarrowtail \sigma\,(\mathrm{S}\,n)) \otimes (h\,(\mathrm{S}\,n) \otimes \text{monoid-fold}'\,n\,(h \circ \mathrm{inc} \circ\ \rightarrowtail \text{trans-perm}\,n\,(\rightarrowtail \sigma\,(\mathrm{S}\,n)) \circ \mathrm{inc} \circ\ \rightarrowtail \sigma'))\\
&\qquad (\textit{claim})\\
&= h\,(\rightarrowtail \sigma\,(\mathrm{S}\,n)) \otimes (h\,(\mathrm{S}\,n) \otimes \text{monoid-fold}'\,n\,(h \circ\ \rightarrowtail \sigma \circ \mathrm{inc} \circ\ \leftarrowtail \text{trans-perm}\,n\,(\leftarrowtail \sigma\,(\mathrm{S}\,n)) \circ \mathrm{inc}))\\
&\equiv h\,(\rightarrowtail \sigma\,(\mathrm{S}\,n)) \otimes \text{monoid-fold}'\,(\mathrm{S}\,n)\,(h \circ\ \rightarrowtail \sigma \circ \mathrm{inc} \circ\ \leftarrowtail \text{trans-perm}\,n\,(\leftarrowtail \sigma\,(\mathrm{S}\,n)))\\
&\qquad \text{(induction)}\\
&= h\,(\rightarrowtail \sigma\,(\mathrm{S}\,n)) \otimes \text{monoid-fold}'\,(\mathrm{S}\,n)\,(h \circ\ \rightarrowtail \sigma \circ \mathrm{inc})\\
&\equiv \text{monoid-fold}'\,(\mathrm{S}\,(\mathrm{S}\,n))\,(h \circ\ \rightarrowtail \sigma)
\end{aligned}
$$

so the aim is to construct a σ' such that the *claim* is true.

First consider the following diagram

$$\begin{tikzcd}
\mathrm{Fin}\,(\mathrm{S}\,n) \arrow[r, "\sigma"] & \mathrm{Fin}\,(\mathrm{S}\,n) \\
\mathrm{Fin}\,n \arrow[u, "\mathrm{inc}"] \arrow[r, "{\downarrow (\text{trans-perm}\,(\rightarrowtail \sigma\,n)\,n \circ \sigma)}"'] & \mathrm{Fin}\,n \arrow[u, "{\text{trans-perm}\,(\rightarrowtail \sigma\,n)\,n \circ \mathrm{inc}}"']
\end{tikzcd}$$

which commutes, because taking some arbitrary m, if $\rightarrowtail\ \sigma\,m \equiv n$ then following the diagram we get n both ways, and otherwise the trans-perms both act as identity and we get $\rightarrowtail \sigma\,m$. We call the construction on the bottom arrow σ^*.

Next consider the following diagram

$$\begin{tikzcd}
\mathrm{Fin}\,n \arrow[r, "\mathrm{inc}"] \arrow[d, "(\sigma^{*} \circ \mathrm{trans\text{-}perm}\,n\,(\leftarrowtail\ \sigma\,(\mathrm{S}\,n)))^{*}"] & \mathrm{Fin}\,(\mathrm{S}\,n) \arrow[rr, "\mathrm{trans\text{-}perm}\,n\,(\leftarrowtail\ \sigma\,(\mathrm{S}\,n))"] & & \mathrm{Fin}\,(\mathrm{S}\,n) \arrow[r, "\mathrm{inc}"] \arrow[dl, "\sigma^{*}"] & \mathrm{Fin}\,(\mathrm{S}\,(\mathrm{S}\,n)) \arrow[d, "\sigma"] \\
\mathrm{Fin}\,n \arrow[rr, "\mathrm{trans\text{-}perm}\,n\,(\rightarrowtail\ \sigma\,(\mathrm{S}\,n)) \circ \mathrm{inc}"'] \arrow[dr, "\mathrm{inc}"'] & & \mathrm{Fin}\,(\mathrm{S}\,n) \arrow[rr, "\mathrm{trans\text{-}perm}\,(\mathrm{S}\,n)\,(\rightarrowtail\ \sigma\,(\mathrm{S}\,n)) \circ \mathrm{inc}"'] & & \mathrm{Fin}\,(\mathrm{S}\,(\mathrm{S}\,n)) \\
& \mathrm{Fin}\,(\mathrm{S}\,n) \arrow[rr, "\mathrm{trans\text{-}perm}\,n\,(\rightarrowtail\ \sigma\,(\mathrm{S}\,n))"'] & & \mathrm{Fin}\,(\mathrm{S}\,n) \arrow[ur, "\mathrm{inc}"'] &
\end{tikzcd}$$

The top line and σ form the right-hand side of the claim and the bottom three arrow the left-hand side. Therefore the $\mathrm{Fin}\,n \to \mathrm{Fin}\,n$ on the left represents the constructed σ'. The top right quadrangle of this diagram is an instance of the σ^{*} construction above. The top left is another instance, which gives us the σ' we need. Finally we need to show that the bottom quadrangle commutes. Take an arbitrary m and assume $m \equiv\ \rightarrowtail\ \sigma\,(\mathrm{S}\,n)$. Along the top this gives n. Along the bottom it also gives n. Now assume $m \not\equiv\ \rightarrowtail\ \sigma\,(\mathrm{S}\,n)$, in which case m is unaffected by all the trans-perms, so in both cases we get m.

Coherence Proving that the proof of monoid-fold is coherent turned out to be extremely difficult, even on paper, so has not been formalised. However, we believe that it should indeed hold, so chose to postulate the result in Agda.

Proofs of Universality We again need to show that this definition satisfies the necessary conditions to be the free construction as outlined in Section 5.4. The unit law follows by computation as

$$\begin{aligned}\mathrm{monoid\text{-}fold}\ f\ \ \bot\text{-}\mathrm{rec}\,(\leftarrowtail\ \mathrm{Fin0\text{-}}\bot) &\equiv \mathrm{monoid\text{-}fold}'\ 0\ f\ (\bot\text{-}\mathrm{rec} \circ\!\leftarrowtail\ \mathrm{Fin0\text{-}}\bot)\\ &\equiv e\end{aligned}$$

The singleton law follows similarly, but requires a call to the right unit law.

The multiplication law is more complex. It has type

$$\Pi_{(x,y:\mathrm{SM}\,C)}\ f^{\#}\,(x \cup y) = f^{\#}\,x \otimes f^{\#}\,y$$

which takes two SM C values. We would like to perform induction on the size of x. The base case is provable because any SM C of size 0 is equivalent to $\varnothing$ and then the left unit law applies. However, the inductive case doesn't work out. This is because we cannot split an arbitrary SM C of size S n to perform induction on the smaller component.

Instead of induction on the size of x we need a new tactic. Intuitively it should be the case that for some property P, if P holds for all Fin n then P should hold for all I

equivalent to some $\mathrm{Fin}\, n$. We want some elimination princple for $\mathrm{SM}\, C$ that allows us to only consider the case where I is a $\mathrm{Fin}\, n$. We can construct such a principle using a lemma from [5]. The lemma states that for types A and B and a depedent type C over B, for any surjective function $f : A \to B$ and any function $g : \Pi_{(a:A)}\, C\,(f\ a)$ such that

$$\text{g-is-const} : \Pi_{(a,a':A)}\Pi_{(p:f\ a=f\ a')}\, g\ a = g\ a'\ [C \downarrow p]$$

there exists a function $h : \Pi_{(b:B)}\, C\ b$ such that $\Pi_{(a:A)}\, h\,(f\ a) = g\ a$.

So consider the function

$$\begin{aligned}&\text{SM-Fin} : \Sigma_{(n:\mathbb{N})}\,(\mathrm{Fin}\, n \to C) \to \mathrm{SM}\, C\\ &\text{SM-Fin}\,(n, f) = (\mathrm{Fin}\, n, f, n, |\mathrm{ide}\,(\mathrm{Fin}\, n)|)\end{aligned}$$

This is clearly surjective, which means we can use the lemma above to write an elimination principle. The principle says that to show a function for all $\mathrm{SM}\, C$ it is enough to show it for all $\text{SM-Fin}\,(n, f)$ and show that it is constant.

Now, for example, to show the multiplication law for the universal property we can use the elimination rule twice, so we need to show

$$\begin{aligned}&\Pi_{(m,n:\mathbb{N})}\Pi_{(i:\mathrm{Fin}\, m\to C)}\Pi_{(j:\mathrm{Fin}\, n\to C)}\\ &f^{\#}\,(\text{SM-Fin}\,(m,i) \cup \text{SM-Fin}\,(n,j)) = f^{\#}\,(\text{SM-Fin}\,(m,i)) \otimes f^{\#}\,(\text{SM-Fin}\,(n,j))\end{aligned}$$

which we can prove by induction because we have concrete values for the truncated equivalences. Then we need to show that this context is constant, which we have not managed to show yet.

6.4 Agda Formalisation

The truncated SM defines all the concepts from Section 5.6.3, therefore the following listings contain only auxilliary concepts used in defining the core ones.

6.4.1 Imposed Finiteness

Concept	Agda Name
SM Truncation level	`SM-level`
Fin0-⊥	`Fin0-⊥`
Fin1-⊤	`Fin1-⊤`
Coproduct Unit Left	`⊔-unit-l`
Coproduct Unit Right	`⊔-unit-r`
Coproduct Associativity	`⊔-assoc`
Coproduct Comm	`⊔-comm`

6.4.2 Functions from Propositional Truncations

Concept	Agda Name
Universal Property for Sets	`Trunc-to-set-equiv`
Universal Property for Groupoids	`Trunc-to-gpd-equiv`

6.4.3 The Universal Property

Concept	Agda Name
Transpose Permutation	`trans-perm`
Transpose Perm Identity	`trans-perm-id`
Transpose Perm Symmetric	`trans-perm-sym`
Transpose Perm Applied to Left Value	`trans-perm-->-l`
Transpose Perm Applied to Right Value	`trans-perm-->-r`
Transpose Perm Self Inverse	`trans-perm-double-idf`
$\downarrow$	`perm-down`
Inc Composed with $\downarrow$	`perm-down-inc`
σ^* Construction	`perm-down-*`
Permutation Invariance of $\mathsf{monoid\text{-}fold}'$	`monoid-fold'-is-perm-invariant`
Finite **SM** Construction	`SM-Fin`
Finite **SM** Construction Surjective	`SM-Fin-surj`
Splitting Finite **SM** Construction	`SM-Fin-S-split`
SM Elimination Principle	`SM-elim`

This formalisation turned out to require almost identical coherence conditions to the **hGpd** model, so again these were postulated.

7 Generalised Species of Structures

Generalised species of structures [4] are defined as

$$C \rightsquigarrow D :\equiv \mathrm{SM}\, C \to \widehat{D},$$

for types C and D. Taking C and D as **hGpd**s, species are the morphisms of the category **Esp**. The proofs of the categorical structure are simpler to define in the opposite category $\mathbf{Esp}^{\mathrm{op}}$ so we will begin there.

7.1 The Category $\mathbf{Esp}^{\mathrm{op}}$

The category $\mathbf{Esp}^{\mathrm{op}}$ has objects as **hGpd**s and homs as functors

$$C \leftsquigarrow D :\equiv C \to \widehat{\mathrm{SM}\, D}$$

between **hGpd**s.

The identity morphism for an **hGpd**, C, is given by

$$\mathrm{id}_C :\equiv Y \circ [\_]$$

combining the singleton of SM and the Yoneda functor.

Composition of morphisms $G : D \leftsquigarrow E$ and $F : C \leftsquigarrow D$ is defined as

$$G \circ F :\equiv \mathrm{Lan}_Y\, G^\# \circ F.$$

To understand this, consider the following diagram

$$\begin{tikzcd}
 & & & C \arrow[dl, "F"] \\
D \arrow[r, "{[\_]}"] \arrow[dr, "G"'] & \mathrm{SM}\, D \arrow[r, "Y"] \arrow[d, "G^\#"] & \widehat{\mathrm{SM}\, D} \arrow[dl, "\mathrm{Lan}_Y\, G^\#"] & \\
 & \widehat{\mathrm{SM}\, E} & &
\end{tikzcd}$$

The left-hand triangle is an application of the universal property for SM and the right-hand triangle is an application of the left Kan extension for the Yoneda functor. Composing the resulting morphism with F gives us a morphism of type $C \to \widehat{\mathrm{SM}\,E}$ as needed.

For the left unit law consider the diagram

$$\begin{tikzcd}
 & & & C \arrow[dl, "F"] \\
D \arrow[r, "{[\_]}"] \arrow[d, "{[\_]}"'] & \mathrm{SM}\,D \arrow[r, "Y"] \arrow[d, "(Y \circ [\_])^{\#}"'] & \widehat{\mathrm{SM}\,D} \arrow[dl, "\mathrm{Lan}_Y (Y \circ [\_])^{\#}"] & \\
\mathrm{SM}\,D \arrow[r, "Y"'] & \widehat{\mathrm{SM}\,D} & &
\end{tikzcd}$$

We have for a monoid homomorphism $h : D \to E$ and a function $f : C \to D$ that

$$(h \circ f)^{\#} = h \circ f^{\#}.$$

The Yoneda functor is a homomorphism, as outlined in Section 4.4. We also have that the universal property applied to the singleton gives the identity function, so therefore

$$(Y \circ [\_])^{\#} = Y.$$

Finally $\mathrm{Lan}_Y\, Y = \mathrm{id}$, so $\mathrm{Lan}_Y\, Y \circ F = F$.

The right unit law and associativity follow from similar diagrams and applications of lemmas about Lans and the universal property, so we have a category.

We can now derive the category **Esp** by taking the dual of $\mathbf{Esp}^{\mathrm{op}}$. We want to show Cartesian closure of the category **Esp**, but to do this requires an equivalence that we will look at first.

7.2 An Important Equivalence

To show that species form a Cartesian Closed Category we will need the following equivalence

$$\mathrm{SM}\,(C \sqcup D) \simeq \mathrm{SM}\,C \times \mathrm{SM}\,D. \tag{7.1}$$

To show this we need to exhibit a function in each direction and show that both compositions are equal to the identity.

The function in the first direction has type $\mathrm{SM}\,(C \sqcup D) \to \mathrm{SM}\,C \times \mathrm{SM}\,D$, so we can use the universal property for SM. Therefore we want to define a function $f : C \sqcup D \to$

$\mathrm{SM}\, C \times \mathrm{SM}\, D$. Such a function can be defined as

$$\begin{aligned} f(\mathrm{inl}\ \ c) &:\equiv ([c], \varnothing) \\ f(\mathrm{inr}\ \ d) &:\equiv (\varnothing, [d]) \end{aligned}$$

and the first direction becomes $f^{\#}$.

In the opposite direction the function has type $\mathrm{SM}\, C \times \mathrm{SM}\, D \to \mathrm{SM}\, (C \sqcup D)$, so we cannot use the universal property directly. Instead, we can apply the functorial action of SM to the injection functions, to get

$$\mathrm{SM}\,\mathrm{inl} \,:\, \mathrm{SM}\, C \to \mathrm{SM}\ (C \sqcup D)$$

and

$$\mathrm{SM}\,\mathrm{inr} \,:\, \mathrm{SM}\, D \to \mathrm{SM}\ (C \sqcup D).$$

These can be applied to the left and right components respectively and the results joined with the union operation. The resulting function is

$$g\ (c, d) :\equiv \mathrm{SM}\,\mathrm{inl}\, c \cup \mathrm{SM}\,\mathrm{inr}\, d.$$

For the first composition consider the following diagram

$$\begin{tikzcd}
\mathrm{SM}\, C \times \mathrm{SM}\, D \arrow[r, "\mathrm{SM}\,\mathrm{inl} \times \mathrm{SM}\,\mathrm{inr}"] \arrow[dr, "(f^{\#} \circ \mathrm{SM}\,\mathrm{inl}) \times (f^{\#} \circ \mathrm{SM}\,\mathrm{inr})"'] & \mathrm{SM}\,(C \sqcup D)^2 \arrow[r, "\cup"] \arrow[d, "f^{\#} \times f^{\#}"] & \mathrm{SM}\,(C \sqcup D) \arrow[r, "f^{\#}"] & \mathrm{SM}\, C \times \mathrm{SM}\, D \arrow[d, "\pi_1"] \\
 & (\mathrm{SM}\, C \times \mathrm{SM}\, D)^2 \arrow[r, "\pi_1 \times \pi_1"'] \arrow[urr, "\cup_{\mathrm{pt}}"] & \mathrm{SM}\, C \times \mathrm{SM}\, C \arrow[r, "\cup"'] & \mathrm{SM}\, C
\end{tikzcd}$$

The top three arrows give the composition in question. We first show that this diagram commutes. Then, if the bottom three arrows give π_1 and in the same diagram with π_2s give π_2, then the whole composition is equal to the identity.

The left-hand triangle commutes by definition. The upper right-hand triangle commutes because the universal extension of f is a monoid homomorphism. $\cup$ is the associative operation of the monoid on SM. $\cup_{\mathrm{pt}}$ applies this operation pointwise, which is the extension of the monoid structure to a pair of monoids. The lower right-hand triangle commutes because π_1 is a monoid homomorphism.

To show that the bottom three arrows give π_1 we need a couple of things. Firstly consider the following diagram

$$\begin{tikzcd}
C \arrow[r, "{[\_]}"] \arrow[d, "f"'] & \mathrm{SM}\,C \arrow[d, "\mathrm{SM}\,f"] \arrow[dd, bend left=60, "(g \circ f)^\#"] \\
D \arrow[r, "{[\_]}"] \arrow[dr, "g"'] & \mathrm{SM}\,D \arrow[d, "g^\#"] \\
 & M
\end{tikzcd}$$

Clearly the composition of two monoid homomorphism is a monoid homomorphism. We can therefore use the uniqueness of the universal property to get

$$g^\# \circ \mathrm{SM}\,f = (g \circ f)^\#.$$

Secondly for two functions $f : C \to D$ and $g : C \to E$ we have

$$\langle f, g \rangle^\# = \langle f^\#, g^\# \rangle,$$

again as a consequence of uniqueness.

Using these, for a pair $(c, d) : \mathrm{SM}\,C \times \mathrm{SM}\,D$ we get

$$\begin{aligned}
&\pi_1(f^\#(\mathrm{SM}\,\mathrm{inl}\,c)) \cup \pi_1(f^\#(\mathrm{SM}\,\mathrm{inr}\,d)) \\
&= \pi_1((f \circ \mathrm{inl})^\#\ c) \cup \pi_1((f \circ \mathrm{inr})^\#\ d) && \text{(first property above)} \\
&\equiv \pi_1((\lambda\ c \to ([c], \varnothing))^\#\ c) \cup \pi_1((\lambda\ d \to (\varnothing, [d]))^\#\ d) \\
&= \pi_1(\langle [\_]^\#, (\mathrm{cst}\ \varnothing)^\# \rangle\ c) \cup \pi_1(\langle (\mathrm{cst}\ \varnothing)^\#, [\_]^\# \rangle\ d) && \text{(second property above)} \\
&= \pi_1(c, \varnothing) \cup \pi_1(\varnothing, d) && \text{(universal property for [\_] and cst } \varnothing) \\
&\equiv c \cup \varnothing \\
&= c && \text{(right unit law)} \\
&\equiv \pi_1(c, d)
\end{aligned}$$

as required, and get a similar result for π_2.

For the second composition we show that $g \circ f^\#$ is a monoid homomorphism and that

$$g\ (f\ [c]) = [c].$$

This implies that $g \circ f = [\_]^\#$ by uniqueness of the universal property. We have that $[\_]^\# = \mathrm{id}$, so $g \circ f = id$.

7.3 Cartesian Closure of Esp

The terminal object of **Esp** is an **hGpd**, $\top_1$, such that for any other **hGpd**, C, there is a unique morphism $C \rightsquigarrow \top_1$. A morphism $C \rightsquigarrow \top_1$ is a function

$$\top_1 \to \mathrm{SM}\,C \to \mathcal{U}$$

so we need a $\top_1$ such that the function space $\top_1 \to \boldsymbol{D}$ is contractible for all $\boldsymbol{D}$. This is satisfied by $\bot$, for which $\bot$-rec is the unique function from $\bot$ to any other type.

The product of two **hGpd**s in **Esp** is given by the coproduct, $\sqcup$. That this is a categorical product is proved using the equivalence

$$C \rightsquigarrow D \sqcup E \simeq (C \rightsquigarrow D) \times (C \rightsquigarrow E)$$

which is simple to define. The forward direction is defined by

$$f\ h :\equiv ((\lambda\, d\, c \to h\,(\mathrm{inl}\, d)\, c), (\lambda\, e\, c \to h\,(\mathrm{inr}\, e)\, c))$$

and the backward direction by

$$\begin{aligned} g\,(h_1, h_2)\,(\mathrm{inl}\, d) &:\equiv h_1\, d \\ g\,(h_1, h_2)\,(\mathrm{inr}\, e) &:\equiv h_2\, e \end{aligned}$$

and these can easily be seen to be inverses.

The exponential object $\boldsymbol{C} \Rightarrow \boldsymbol{D}$ is defined as

$$\boldsymbol{C} \Rightarrow \boldsymbol{D} :\equiv \mathrm{SM}\, \boldsymbol{C} \times \boldsymbol{D}.$$

That this is indeed an exponential is proved using the equivalence

$$\boldsymbol{C} \sqcup \boldsymbol{D} \rightsquigarrow \boldsymbol{E} \simeq \boldsymbol{C} \rightsquigarrow \boldsymbol{D} \Rightarrow \boldsymbol{E}.$$

We have

$$\begin{aligned} \boldsymbol{C} \sqcup \boldsymbol{D} \rightsquigarrow \boldsymbol{E} &\equiv \boldsymbol{E} \to \widehat{\mathrm{SM}\,(\boldsymbol{C} \sqcup \boldsymbol{D})} \\ &\simeq \boldsymbol{E} \to \widehat{\mathrm{SM}\,\boldsymbol{C} \times \mathrm{SM}\,\boldsymbol{D}} && \text{(Equation (7.1))} \\ &\simeq \mathrm{SM}\,\boldsymbol{D} \times \boldsymbol{E} \to \widehat{\mathrm{SM}\,\boldsymbol{C}} && \text{(Currying, reordering, uncurrying)} \\ &\equiv \boldsymbol{C} \rightsquigarrow \boldsymbol{D} \Rightarrow \boldsymbol{E} \end{aligned}$$

7.4 The Differential Calculus of Species

We now define the differential calculus of species [3], a set of operations on species that obey rules analagous to those of analysis.

Addition The $\boxplus$ operation behaves like addition. It is a binary operation that takes two species of the same type and returns a third. For two presheaves $\boldsymbol{P}, \boldsymbol{Q} : \boldsymbol{C} \rightsquigarrow \boldsymbol{D}$, the definition is

$$(\boldsymbol{P} \boxplus \boldsymbol{Q})\, d\, m :\equiv (\boldsymbol{P}\, d\, m) \sqcup (\boldsymbol{Q}\, d\, m).$$

The unit for this operation is

$$0\, d\, m \;:\equiv \bot$$

which satisifies left and right unit laws. They hold because we have

$$\bot \sqcup C \simeq C$$

and

$$C \sqcup \bot \simeq C$$

which we used when defining the union of the truncated version of SM. This operation is also commutative and associative, again from the commutativity and associativity of $\sqcup$. This also distributes with species composition as

$$\Pi_{(P : C \rightsquigarrow D)} \Pi_{(Q, R : D \rightsquigarrow E)}\, (Q \boxplus R) \circ P = (Q \circ P) \boxplus (R \circ P),$$

which follows from the action of Lan_Y on coproducts.

Multiplication The $\boxtimes$ operation behaves like multiplication. For two species $P, Q : C \rightsquigarrow D$, the definition is

$$(P \boxtimes Q)\, d \;:\equiv (P\, d \,\hat{\otimes}\, Q\, d).$$

The unit for this operation is

$$1\, d \;:\equiv \hat{I},$$

which satisifies the unit laws because of the unit laws for $\hat{\otimes}$. This operation is commutative and associative, again from $\hat{\otimes}$. It also distributes with species composition as

$$\Pi_{(P : C \rightsquigarrow D)} \Pi_{(Q, R : D \rightsquigarrow E)}\, (Q \boxtimes R) \circ P = (Q \circ P) \boxtimes (R \circ P),$$

which follows from $\mathrm{Lan}_Y\, P^{\#}$ being a monoid homomorphism.

Differentiation The ∂ operation behaves as a partial derivative. Given an $c : C$ and a species $P : C \rightsquigarrow D$ it gives a new $C \rightsquigarrow D$. The definition is

$$\partial\, c\, P\, d\, m \;:\equiv P\, d\, (m \cup [c]).$$

For intuition about this operation, take C and D to be $\top$. Now $\mathrm{SM}\,\top \simeq \mathbb{N}$ and $(\top \to C) \simeq C$ so we can treat P as having type $\mathbb{N} \to \mathcal{U}$. We interpret P_n as the coefficients of a power series

$$p(x) = \sum_{n \geq 0} P_n \frac{x^n}{n!}$$

which differentiates to

$$p'(x) = \sum_{n \geq 0} P_n \frac{x^{n-1}}{(n-1)!}.$$

We can see this as

$$p'(x) = \sum_{i \geq 0} P_{i+1} \frac{x^i}{i!},$$

so differentiation shifts the index by 1. So now for an arbitrary $\boldsymbol{C}$, we can shift in any direction c by adding 1 value of this element to the multiset.

This operation satisifies some commutativity property

$$\Pi_{(c,c' : C)} \, \partial \, c \, (\partial \, c' \, \boldsymbol{P}) = \partial \, c' \, (\partial \, c \, \boldsymbol{P}),$$

which holds by associativity and commutativity of $\cup$. It also satisfies

$$\Pi_{(c : C)} \Pi_{(P,Q : C \rightsquigarrow D)} \, \partial \, c \, (\boldsymbol{P} \boxplus \boldsymbol{Q}) = \partial \, c \, \boldsymbol{P} \boxplus \partial \, c \, \boldsymbol{Q},$$

and

$$\Pi_{(c : C)} \, \partial \, c \, \boldsymbol{E} = \boldsymbol{E},$$

where $\boldsymbol{E}$ is the constantly $\top$ presheaf of two arguments, named for its analogy to Euler's number, e. Both of these hold judgementally.

We can also define the operation δ, which given a species $\boldsymbol{C} \rightsquigarrow \boldsymbol{D}$ returns a species $\boldsymbol{C} \rightsquigarrow \boldsymbol{C} \times \boldsymbol{D}$ and is defined by

$$\delta \, \boldsymbol{P} \, (c, d) \, m \; :\equiv \partial \, c \, \boldsymbol{P} \, d \, m.$$

This corresponds to the Jacobian matrix which for some $f : \mathbb{R}^n \to \mathbb{R}^m$ is an $m \times n$ matrix of partial derivatives. Given some $j : \mathbb{R}^n$ we get a linear operator from $\mathbb{R}^n$ to $\mathbb{R}^m$ by taking the appropriate column of the matrix. This map

$$\mathbb{R}^n \to \mathrm{Lin}(\mathbb{R}^n, \mathbb{R}^m)$$

corresponds to our

$$\boldsymbol{C} \rightsquigarrow \boldsymbol{C} \times \boldsymbol{D}.$$

With these operations we can prove the Leibniz Rule

$$\partial_c \, (\boldsymbol{P} \cdot \boldsymbol{Q}) = \partial_c(\boldsymbol{P}) \cdot \boldsymbol{Q} + \boldsymbol{P} \cdot \partial_c(\boldsymbol{Q})$$

which translates to

$$\partial \, c \, (\boldsymbol{P} \boxtimes \boldsymbol{Q}) = (\partial \, c \, \boldsymbol{P} \boxtimes \boldsymbol{Q}) \boxplus (\boldsymbol{P} \boxtimes \partial \, c \, \boldsymbol{Q}).$$

We have

$$
\begin{aligned}
& \partial\, c\, (P \boxtimes Q)\, d\, m \\
\equiv\ & (P \boxtimes Q)\, d\, (m \cup [c]) \\
\equiv\ & \Sigma_{(m_1, m_2 : \mathrm{SM}\, C)}\, P\, d\, m_1 \times Q\, d\, m_2 \times (m \cup [c] = m_1 \cup m_2) \\
=\ & \Sigma_{(m_1, m_2 : \mathrm{SM}\, C)}\, P\, d\, m_1 \times Q\, d\, m_2 \\
& \qquad\qquad \times ((\Sigma_{(m' : \mathrm{SM}\, C)}\, (m = m' \cup m_2) \times (m' \cup [c] = m_1)) \\
& \qquad\qquad\quad \sqcup \\
& \qquad\qquad\quad (\Sigma_{(m' : \mathrm{SM}\, C)}\, (m = m_1 \cup m') \times (m' \cup [c] = m_2))) && \text{(combinatorial lemma)} \\
=\ & (\Sigma_{(m_1, m_2 : \mathrm{SM}\, C)}\, P\, d\, m_1 \times Q\, d\, m_2 \\
& \qquad\qquad \times \Sigma_{(m' : \mathrm{SM}\, C)}\, (m = m' \cup m_2) \times (m' \cup [c] = m_1)) \\
& \sqcup \\
& (\Sigma_{(m_1, m_2 : \mathrm{SM}\, C)}\, P\, d\, m_1 \times Q\, d\, m_2 \\
& \qquad\qquad \times \Sigma_{(m' : \mathrm{SM}\, C)}\, (m = m_1 \cup m') \times (m' \cup [c] = m_2)) \\
=\ & (\Sigma_{(m', m_2 : \mathrm{SM}\, C)}\, P\, d\, (m' \cup [c]) \times Q\, d\, m_2 \times (m = m' \cup m_2)) \\
& \sqcup \\
& (\Sigma_{(m_1, m' : \mathrm{SM}\, C)}\, P\, d\, m_1 \times Q\, d\, (m' \cup [c]) \times (m = m_1 \cup m')) && \text{(density formula twice)} \\
\equiv\ & (\partial\, c\, P \boxtimes Q) \boxplus (P \boxtimes \partial\, c\, Q)
\end{aligned}
$$

7.5 Agda Formalisation

7.5.1 The Category $\mathbf{Esp}^{\mathrm{op}}$

Concept	Agda Name
↜	`_ ↜ _`
Identities	`id`$_2$
Composition	`_ ∘`$_2$` _`
Composition Unit Left	`_ ∘`$_2$` _-unit-l`
Composition Unit Right	`_ ∘`$_2$` _-unit-r`
Composition Associativity	`_ ∘`$_2$` _-assoc`
Composition Commutativity	`_ ∘`$_2$` _-comm`
Universal Extension of Composition	`lan-y-∘-up`

7.5.2 An Important Equivalence

Concept	Agda Name
π_1 Homomorphism Mult	`fst-hom-mult`
π_2 Homomorphism Mult	`snd-hom-mult`
Universal Extension on Pairs	`up-pair`
SM Functor Composed with Universal Property	`SM-fmap-up`
SM Coproduct/Product Equivalence	`SM-⊔-×-equiv`

7.5.3 Cartesian Closure of **Esp**

Concept	Agda Name
Product in **Esp**	$\_\times_s\_$
Product Equivalence	$\times_s$`-equiv`
Exponential in **Esp**	$\_\Rightarrow_s\_$
Exponential Equivalence	$\Rightarrow_s$`-equiv`

7.5.4 The Differential Calculus of Species

Concept	Agda Name
Addition	$\_\boxplus\_$
0	0_s
Addition Unit Left	$\boxplus$`-unit-l`
Addition Unit Right	$\boxplus$`-unit-r`
Addition Unit Associativity	$\boxplus$`-assoc`
Addition Unit Commutativity	$\boxplus$`-comm`
Addition Distributivity over Composition	$\boxplus$-$\circ_s$`-l`
Multiplication	$\_\boxtimes\_$
1	1_s
Multiplication Unit Left	$\boxtimes$`-unit-l`
Multiplication Unit Right	$\boxtimes$`-unit-r`
Multiplication Unit Associativity	$\boxtimes$`-assoc`
Multiplication Unit Commutativity	$\boxtimes$`-comm`
Multiplication Distributivity over Composition	$\boxtimes$-$\circ_s$`-l`
Partial Differentiation	∂
Partial Diff Commutativity	∂`-comm`
Partial Diff of Addition	∂-$\boxplus$
Partial Diff of *E*	∂`-E`
Leibniz Rule	`leibniz`
Jacobian	δ

The combinatorial lemma itself was left postulated.

8 Summary & Conclusions

Bibliography

[1] F. Bergeron, G. Labelle, and P. Leroux. *Combinatorial species and tree-like structures*. Number 67 in Encyclopedia of Mathematics and its Applications. Cambridge University Press, Cambridge, 1998.

[2] Guillaume Brunerie, Kuen-Bang Hou (Favonia), Evan Cavallo, Eric Finster, Jesper Cockx, Christian Sattler, Chris Jeris, Michael Shulman, et al. Homotopy type theory in Agda. URL `https://github.com/HoTT/HoTT-Agda`.

[3] Marcelo Fiore. Mathematical models of computational and combinatorial structures. 2005.

[4] Marcelo Fiore, Nicola Gambino, Martin Hyland, and Glynn Winskel. The cartesian closed bicategory of generalised species of structures. *Journal of the London Mathematical Society*, 77(1):203–220, 2008.

[5] Kuen-Bang Hou. *Higher-Dimensional Types in the Mechanization of Homotopy Theory*. PhD thesis, Carnegie Mellon University, 2017.

[6] André Joyal. Une théorie combinatoire des Séries formelles. *Advances in Mathematics*, 42(1):1–82, 1981.

[7] André Joyal. Foncteurs analytiques et especes de structures. In *Combinatoire énumérative*, volume 1234 of *Lecture Notes in Mathematics*, pages 126–159. Springer Verlag, 1986.

[8] Nicolai Kraus. The general universal property of the propositional truncation. *Postproceedings of Types for Proofs and Programs (TYPES)*, 2014.

[9] Per Martin-Löf. An intuitionistic theory of types: Predicative part. *Studies in Logic and the Foundations of Mathematics*, 80:73–118, 1975.

[10] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

[11] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Founda-*

tions of Mathematics. `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

[12] Brent Abraham Yorgey. *Combinatorial species and labelled structures*. PhD thesis, University of Pennsylvania, 2014.