Symmetries in Sorting

Anonymous Author(s)

Abstract

Sorting algorithms are fundamental to computer science, and their correctness criteria are well understood as rearranging elements of a list according to a specified total order on the underlying set of elements. Unfortunately, as mathematical functions, they are rather violent, because they perform combinatorial operations on the representation of the input list. In this paper, we study sorting algorithms conceptually as abstract sorting functions. We show that sorting functions determine a well-behaved section (right inverse) to the canonical surjection sending a free monoid to a free commutative monoid of its elements. Introducing symmetry by passing from free monoids (ordered lists) to free commutative monoids (unordered lists) eliminates ordering, while sorting (the right inverse) recovers ordering. From this, we give an axiomatization of sorting which does not require a pre-existing total order on the underlying set, and then show that there is an equivalence between (decidable) total orders on the underlying set and correct sorting functions.

The first part of the paper develops concepts from universal algebra from the point of view of functorial signatures, and gives various constructions of free monoids and free commutative monoids in type theory, which are used to develop the second part of the paper about the axiomatization of sorting functions. The paper uses informal mathematical language, and comes with an accompanying formalization in Cubical Agda.

Keywords: universal algebra, category theory, type theory, homotopy type theory, combinatorics, formalization

1 Introduction

Consider a puzzle about sorting, inspired by Dijkstra's Dutch National Flag problem [Dijkstra 1997, Ch.14]. Suppose there are balls of three colors, corresponding to the colors of the Dutch flag: red, white, and blue.

{🔴, ⚪, 🔵}

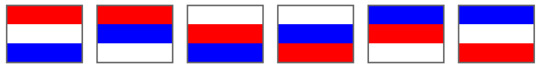
Given an unordered list (bag) of such balls, how many ways can you sort them into the Dutch flag?

⦃🔴, 🔴, 🔵, ⚪, 🔵, 🔴, ⚪, 🔵⦄

Obviously there is only one way, decided by the order the colors appear in the Dutch flag: red < white < blue.

[🔴, 🔴, 🔴, ⚪, ⚪, 🔵, 🔵, 🔵]

What if we are avid enjoyers of vexillology who also want to consider other flags? We might ask: how many ways can we sort our unordered list of balls? We know that there are only $3! = 6$ permutations of $\{\text{red}, \text{white}, \text{blue}\}$, so there are only 6 possible orderings we can define. In fact, there are exactly 6 such categories of tricolor flags (see Wikipedia). We have no allegiance to any of the countries presented by the flags, hypothetical or otherwise – this is purely a matter of combinatorics.



We posit that, because there are exactly 6 orderings, we can only define 6 *extensionally correct* sorting functions. Formally, there is a bijection between the set of orderings on a carrier set A and the set of correct sorting functions on lists of A. In fact, a sorting function can be correctly axiomatized just from the point of view of this bijection!

Outline and Contributions. The paper is organized as follows:

- In § 2, we remark on the notation and type-theoretic conventions used in the paper.
- In § 3, we describe a formalization of universal algebra developed from the point of view of functorial signatures, the definition and universal property of free algebras, and algebras satisfying an equational theory.
- In § 4, we give various constructions of free monoids, and their proofs of universal property. Then, in § 5, we add symmetry to each representation of free monoids, and extend the proofs of universal property from free monoids to free commutative monoids. These constructions are well-known, but we formalize them conceptually by performing formal combinatorial operations.
- In § 6, we build on the constructions of the previous sections and study sorting functions. The main result in this section is to connect total orders, sorting, and symmetry, by proving an equivalence between decidable total orders on a carrier set A, and correct sorting functions on lists of A.
- All the work in this paper is formalized in Cubical Agda, which is discussed in § 7. The accompanying code is available as supplementary material.
- § 8 discusses related and future work.

The three main parts of the paper can be read independently. Readers interested in the formalization of universal algebra can start from § 3. Readers interested in the constructions of free monoids and free commutative monoids can skip ahead to §§ 4 and 5. If the reader already believes in the existence of free algebras for monoids and commutative

CPP'25, Jan 19–25, 2025, Denver, CO, USA
2024. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

monoids, they can directly skip to the application section on sorting, in § 6. Although the formalization is a contribution in itself, the purpose of the paper is not to directly discuss the formalization, but to present the results in un-formalized form (in type-theoretic foundations), so the ideas are accessible to a wider audience.

2 Notation

The text follows the notational conventions of the HoTT/UF book [Univalent Foundations Program 2013]. The work is formalized in Cubical Agda which uses Cubical Type Theory – we refer the readers to other works such as [Vezzosi et al. 2019] for an in-depth tutorial on Cubical Type Theory and programming in Cubical Agda.

We denote the type of types with $\mathcal{U}$, and choose to drop universe levels. We use $\times$ for product types and $+$ for coproduct types. For mere propositions, we use $\land$ to denote conjunction, and $\lor$ to denote logical disjunction (truncated coproduct). We use Fin_n to denote finite sets of cardinality n in HoTT. hProp and hSet denote the universe of propositions and sets, respectively, and we write Set to denote the (univalent) category of sets and functions.

3 Universal Algebra

We first develop some basic notions from universal algebra and equational logic [Birkhoff 1935]. Universal algebra is the abstract study of algebraic structures, which have (algebraic) operations and (universal) equations. This gives us a vocabulary and framework to express our results in. The point of view we take is the standard category-theoretic approach to universal algebra, which predates the Lawvere theory or abstract clone point of view. We keep a running example of monoids in mind, while explaining and defining the abstract concepts.

3.1 Algebras

Definition 3.1 (Signature)**.** A signature, denoted σ, is a (dependent) pair consisting of:

- a set of operations, $\mathsf{op} \colon \mathsf{Set}$,
- an arity function for each symbol, $\mathsf{ar} \colon \mathsf{op} \to \mathsf{Set}$.

Example 3.2. A monoid is a set with an identity element (a nullary operation), and a binary multiplication operation, with signature $\sigma_{\mathsf{Mon}} :\equiv (\mathsf{Fin}_2, \lambda\{0 \mapsto \mathsf{Fin}_0; 1 \mapsto \mathsf{Fin}_2\})$.

Every signature σ induces a signature functor F_σ on Set.

Definition 3.3 (Signature functor $F_\sigma \colon \mathsf{Set} \to \mathsf{Set}$)**.**

$$X \mapsto \textstyle\sum_{(o\,:\,\mathsf{op})} X^{\mathsf{ar}(o)}$$

$$X \xrightarrow{f} Y \mapsto \textstyle\sum_{(o\,:\,\mathsf{op})} X^{\mathsf{ar}(o)} \xrightarrow{(o, - \circ f)} \sum_{(o\,:\,\mathsf{op})} Y^{\mathsf{ar}(o)}$$

Example 3.4. The signature functor for monoids, $F_{\sigma_{\mathsf{Mon}}}$, assigns to a carrier set X, the sets of inputs for each operation. Expanding the dependent product on Fin_2, we obtain a coproduct of sets: $F_{\sigma_{\mathsf{Mon}}}(X) \simeq X^{\mathsf{Fin}_0} + X^{\mathsf{Fin}_2} \simeq \mathbf{1} + (X \times X)$.

A σ-structure is given by a carrier set, with functions interpreting each operation symbol. The signature functor applied to a carrier set gives the inputs to each operation, and the output is simply a map back to the carrier set. Formally, these two pieces of data are an algebra for the F_σ functor. We write $\mathfrak{X}$ for a σ-structure with carrier set X, following the model-theoretic notational convention.

Definition 3.5 (Structure)**.** A σ-structure $\mathfrak{X}$ is an F_σ-algebra, that is, a pair consisting of:

- a carrier set X, and
- an algebra map $\alpha_X \colon F_\sigma(X) \to X$.

Example 3.6. Concretely, an $F_{\sigma_{\mathsf{Mon}}}$-algebra has the type

$$\begin{aligned}\alpha_X \colon F_{\sigma_{\mathsf{Mon}}}(X) \to X &\simeq (\mathbf{1} + (X \times X)) \to X \\ &\simeq (\mathbf{1} \to X) \times (X \times X \to X)\end{aligned}$$

which is the pair of functions interpreting the two operations. Natural numbers $\mathbb{N}$ with $(0, +)$ or $(1, \times)$ are examples of monoid structures.

Definition 3.7 (Homomorphism)**.** A homomorphism between two σ-structures $\mathfrak{X}$ and $\mathfrak{Y}$ is a morphism of F_σ-algebras, that is, a map $f \colon X \to Y$ such that:

$$\begin{tikzcd} F_\sigma(X) \arrow[r, "\alpha_X"] \arrow[d, "F_\sigma(f)"'] & X \arrow[d, "f"] \\ F_\sigma(Y) \arrow[r, "\alpha_Y"'] & Y \end{tikzcd}$$

Example 3.8. Given two monoids $\mathfrak{X}$ and $\mathfrak{Y}$, the top half of the diagram is: $\mathbf{1} + (X \times X) \xrightarrow{\alpha_X} X \xrightarrow{f} Y$, which applies f to the output of each operation, and the bottom half is: $\mathbf{1} + (X \times X) \xrightarrow{F_{\sigma_{\mathsf{Mon}}}(f)} \mathbf{1} + (Y \times Y) \xrightarrow{\alpha_Y} Y$. In other words, a homomorphism between X and Y is a map f on the carrier sets that commutes with the interpretation of the monoid operations, or simply, preserves the monoid structure.

For a fixed signature σ, the category of F_σ-algebras and their morphisms form a category of algebras, written F_σ-Alg, or simply, σ-Alg, given by the obvious definitions of identity and composition of the underlying functions.

3.2 Free Algebras

The category σ-Alg is a category of structured sets and structure-preserving maps, which is an example of a concrete category, that admits a forgetful functor $U \colon \sigma\text{-Alg} \to \mathsf{Set}$, In our notation, $U(\mathfrak{X})$ is simply X, a fact we exploit to simplify our notation, and formalization. The left adjoint to this forgetful functor is the free algebra construction, also known as the term algebra (or the *absolutely free* algebra without equations). We rephrase this in more concrete terms.

Definition 3.9 (Free Algebras)**.** A free σ-algebra construction consists of the following data:

- a set $F(X)$, for every set X,
- a σ-structure on $F(X)$, written as $\mathfrak{F}(X)$,
- a universal map $\eta_X : X \to F(X)$, for every X, such that,
- for any σ-algebra $\mathfrak{Y}$, the operation assigning to each homomorphism $f : \mathfrak{X} \to \mathfrak{Y}$, the map $f \circ \eta_X : X \to Y$ (or, post-composition with η_X), is an equivalence.

More concretely, we are asking for a bijection between the set of homomorphisms from the free algebra to any other algebra, and the set of functions from the carrier set of the free algebra to the carrier set of the other algebra. In other words, there should be no more data in homomorphisms out of the free algebra than there is in functions out of the carrier set, which is the property of *freeness*. The inverse operation to post-composition with η_X is the *universal extension* of a function to a homomorphism,

Definition 3.10 (Universal extension)**.** The universal extension of a function $f : X \to Y$ to a homomorphism out of the free σ-algebra on X is written as $f^\sharp : \mathfrak{F}(X) \to \mathfrak{Y}$. It satisfies the identities: $f^\sharp \circ \eta_X \sim f$, ${\eta_X}^\sharp \sim \mathrm{id}_{\mathfrak{F}(X)}$, and $(g^\sharp \circ f)^\sharp \sim g^\sharp \circ f^\sharp$.

Free algebra constructions are canonically equivalent.

Proposition 1. *Suppose $\mathfrak{F}(X)$ and $\mathfrak{G}(X)$ are both free σ-algebras on X. Then $\mathfrak{F}(X) \simeq \mathfrak{G}(X)$, natural in X.*

So far, we've only discussed abstract properties of free algebras, but not actually given a construction! In type theory, *free* constructions are often given by inductive types, where the constructors are the pieces of data that freely generate the structure, and the type-theoretic induction principle enforces the category-theoretic universal property.

Definition 3.11 (Construction of Free Algebras)**.** The free σ-algebra on a type X is given by the inductive type:

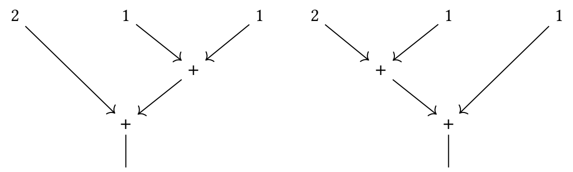
```
data Tree (X : 𝒰) : 𝒰 where
  leaf : X → Tree X
  node : Fσ(Tree X) → Tree X
```

The constructors `leaf` and `node` are, abstractly, the generators for the universal map, and the algebra map, respectively. Concretely, this is the type of abstract syntax trees for terms in the signature σ – the leaves are the free variables, and the nodes are the branching operations of the tree, marked by the operations in σ.

Proposition 2. *`(Tree(X),leaf)` is the free σ-algebra on X.*

3.3 Equations

The algebraic framework of universal algebra we have described so far only captures operations, not equations. These algebras are *lawless* (or *wild* or *absolutely free*) – saying the $F_{\sigma_{\mathsf{Mon}}}$-algebras are monoids, or $\mathfrak{F}_{\sigma_{\mathsf{Mon}}}$-algebras are free monoids is not justified. For example, by associativity, these two trees of $(\mathbb{N}, +)$ should be identified as equal.



To impose equations on the operations, we adopt the point of view of equational logic.

Definition 3.12 (Equational Signature)**.** An equational signature, denoted ε, is a (dependent) pair consisting of:

- a set of names for equations, $\mathsf{eq} : \mathsf{Set}$,
- an arity of free variables for each equation, $\mathsf{fv} : \mathsf{eq} \to \mathsf{Set}$.

Example 3.13. The equational signature for monoids $\varepsilon_{\mathsf{Mon}}$ is: $(\mathsf{Fin}_3, \lambda\{0 \mapsto \mathsf{Fin}_1; 1 \mapsto \mathsf{Fin}_1; 2 \mapsto \mathsf{Fin}_3\})$. The three equations are the left and right unit laws, and the associativity law – a 3-element set of names $\{\mathsf{unitl}, \mathsf{unitr}, \mathsf{assoc}\}$. The two unit laws use one free variable, and the associativity law uses three free variables.

Just like the signature functor Definition 3.3, this produces an equational signature functor on Set.

Definition 3.14 (Eq. Signature Functor F_ε : Set $\to$ Set)**.**

$$X \mapsto \textstyle\sum_{(e:\ \mathsf{eq})} X^{\mathsf{fv}(e)}$$

$$X \xrightarrow{f} Y \mapsto \textstyle\sum_{(e:\ \mathsf{eq})} X^{\mathsf{ar}(e)} \xrightarrow{(e, -\circ f)} \textstyle\sum_{(e:\ \mathsf{eq})} Y^{\mathsf{ar}(e)}$$

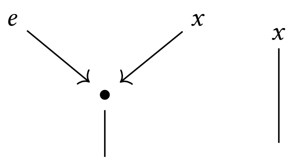
To build equations out of this signature, we use the σ-operations and construct trees for the left and right-hand sides of each equation using the free variables available – a system of equations.

Definition 3.15 (System of Equations)**.** A system of equations over a signature (σ, ε), is a pair of natural transformations:

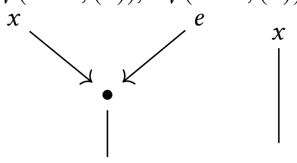
$$\ell, r : F_\varepsilon \Rightarrow \mathfrak{F}_\sigma \ .$$

Concretely, for any set (of variables) V, this gives a pair of trees $\ell_V, r_V : F_\varepsilon(V) \to \mathfrak{F}_\sigma(V)$, and naturality ensures correctness of renaming.

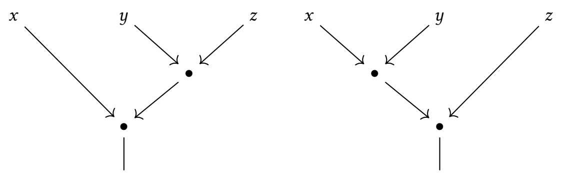
Example 3.16. Given $x : V$, $\ell_V(\mathsf{unitl}, (x))$, $r_V(\mathsf{unitl}, (x))$ are defined as:



Given $x : V$, $\ell_V(\mathsf{unitr}, (x))$, $r_V(\mathsf{unitr}, (x))$ are defined as:



Given $x, y, z : V$, $\ell_V(\mathsf{assocr}, (x, y, z))$, $r_V(\mathsf{assocr}, (x, y, z))$ are defined as:



Finally, we have to say how a given σ-structure $\mathfrak{X}$ *satisfies* the system of equations $T_{(\sigma,\varepsilon)}$. We need to assign a value to each free variable in the equation, picking them out of the carrier set, which is the valuation function $\rho\colon V \longrightarrow X$. Given such an assignment, we evaluate the left and right trees of the equation, by extending ρ (using Definition 3.10), that is by construction, a homomorphism from $\mathfrak{F}(V)$ to $\mathfrak{X}$. To satisfy an equation, these two evaluations should agree.

Definition 3.17 ($\mathfrak{X} \vDash T$)**.** A σ-structure $\mathfrak{X}$ satisfies the system of equations $T_{(\sigma,\varepsilon)}$ if for every set V, and every assignment $\rho\colon V \longrightarrow X$, $\rho^\sharp$ is a (co)fork:

$$\begin{tikzcd} F_\varepsilon(V) \arrow[r, shift left, "\ell_V"] \arrow[r, shift right, "r_V"'] & \mathfrak{F}(V) \arrow[r, "\rho^\sharp"] & \mathfrak{X} \end{tikzcd}$$

There is a full subcategory of σ-Alg which is the variety of algebras satisfying these equations. Constructions of free objects for any arbitrary variety requires non-constructive principles [Blass 1983, § 7, pg.142], in particular, the arity sets need to be projective, so we do not give the general construction. The non-constructive principles can be avoided if we limit ourselves to specific constructions where everything is finitary. Of course, HoTT/UF offers an alternative by allowing higher generators for equations using HITs [Univalent Foundations Program 2013]. We do not develop the framework further, since we have enough tools to develop the next sections.

4 Constructions of Free Monoids

In this section, we consider various constructions of free monoids in type theory, with proofs of their universal property. Since each construction satisfies the same categorical universal property, by Proposition 1, these are canonically equivalent (hence equal, by univalence) as types (and as monoids), allowing us to transport proofs between them. Using the unviersal property allows us to define and prove our programs correct in one go, which is used in § 6.

4.1 Lists

Cons-lists (or sequences) are simple inductive datatypes, well-known to functional programmers, and are the most common representation of free monoids in programming languages. In category theory, these correspond to Kelly's notion of algebraically-free monoids [Kelly 1980].

Definition 4.1 (Lists)**.**

```
data List (A : 𝓤) : 𝓤 where
  [] : List A
  _::_ : A → List A → List A
```

The (universal) generators map is the singleton: $\eta_A(a) :\equiv [a] \equiv$ `a :: []`, the identity element is the empty list `[]`, and the monoid operation $+\!\!+$ is given by concatenation.

Proposition 3. $(-)^\sharp$ *lifts a function* $f\colon A \longrightarrow X$ *to a monoid homomorphism* $f^\sharp\colon \mathsf{List}(A) \longrightarrow \mathfrak{X}$.

Proposition 4 (Universal property for List)**.** $(\mathsf{List}(A), \eta_A)$ *is the free monoid on* A.

4.2 Array

An alternate (non-inductive) representation of the free monoid on a carrier set, or alphabet A, is A^*, the set of all finite words or strings or sequences of characters *drawn* from A, which was known in category theory from [Dubuc 1974]. In computer science, we think of this as an *array*, which is a pair of a natural number n, denoting the length of the array, and a lookup (or index) function $\mathsf{Fin}_\mathsf{n} \longrightarrow A$, mapping each index to an element of A. In type theory, this is also often understood as a container [Abbott et al. 2003], where $\mathbb{N}$ is the type of shapes, and Fin is the type (family) of positions.

Definition 4.2 (Arrays)**.**

```
Array : 𝓤 → 𝓤
Array A = Σ(n : Nat) (Fin n → A)
```

For example, $(3, \lambda\{0 \mapsto 3, 1 \mapsto 1, 2 \mapsto 2\})$ represents the same list as $[3, 1, 2]$. The (universal) generators map is the singleton: $\eta_A(a) = (1, \lambda\{0 \mapsto a\})$, the identity element is $(0, \lambda\{\})$ and the monoid operation $+\!\!+$ is array concatenation.

Lemma 1. *Zero-length arrays* $(0, f)$ *are contractible.*

Definition 4.3 (Concatenation)**.** The concatenation operation $+\!\!+$, is defined below, where $\oplus\colon (\mathsf{Fin}_\mathsf{n} \longrightarrow A) \longrightarrow (\mathsf{Fin}_\mathsf{m} \longrightarrow A) \longrightarrow (\mathsf{Fin}_\mathsf{n+m} \longrightarrow A)$ is a combine operation:

$$(n, f) +\!\!+ (m, g) = (n + m, f \oplus g)$$

$$(f \oplus g)(k) = \begin{cases} f(k) & \text{if } k < n \\ g(k-n) & \text{otherwise} \end{cases}$$

Proposition 5. $(\mathsf{Array}(A), +\!\!+)$ *is a monoid.*

Lemma 2 (Array cons)**.** *Any array* $(S(n), f)$ *is equal to* $\eta_A(f(0)) +\!\!+ (n, f \circ S)$.

Lemma 3 (Array split)**.** *For any array* $(S(n), f)$ *and* (m, g)*,*

$$(n + m, (f \oplus g) \circ S) = (n, f \circ S) +\!\!+ (m, g) \ .$$

Informally, this means given an non-empty array xs and any array ys, concatenating xs with ys then dropping the first element is the same as dropping the first element of xs then concatenating with ys.

Definition 4.4 (Universal extension). Given a monoid $\mathfrak{X}$, and a map $f: A \longrightarrow X$, we define $f^\sharp: \mathsf{Array}(A) \longrightarrow X$, by induction on the length of the array:

$$f^\sharp(0, g) = e$$
$$f^\sharp(S(n), g) = f(g(0)) \bullet f^\sharp(n, g \circ S)$$

Proposition 6. *$(-)^\sharp$ lifts a function $f: A \to X$ to a monoid homomorphism $f^\sharp: \mathsf{Array}(A) \to \mathfrak{X}$.*

Proposition 7 (Universal property for Array)**.** *$(\mathsf{Array}(A), \eta_A)$ is the free monoid on A.*

Proof sketch. We need to show that $(-)^\sharp$ is an inverse to $(-) \circ \eta_A$. $f^\sharp \circ \eta_A = f$ for all set functions $f: A \to X$ holds trivially. To show $(f \circ \eta_A)^\sharp = f$ for all homomorphisms $f: \mathsf{Array}(A) \to \mathfrak{X}$, we need $\forall xs. (f \circ \eta_A)^\sharp(xs) = f(xs)$. Lemmas 2 and 3 allow us to do induction on arrays, therefore we can prove $\forall xs. (f \circ \eta_A)^\sharp(xs) = f(xs)$ by induction on xs, very similarly to how this was proven for List. □

Remark. An alternative proof of the universal property for Array can be given by directly constructing an equivalence (of types, and monoid structures) between $\mathsf{Array}(A)$ and $\mathsf{List}(A)$ (using tabulate and lookup), and then using univalence and transport (see formalization).

5 Constructions of Free Comm. Monoids

The next step is to add commutativity to each construction of free monoids. Informally, adding commutativity to free monoids turns "ordered lists" to "unordered lists", where the ordering is the one imposed by the position or index of the elements in the list. This is crucial to our goal of studying sorting, as we will study sorting as a function mapping back unordered lists to ordered lists, which is later in § 6.3.

It is known that finite multisets are (free) commutative monoids, under the operation of multiset union: $xs \cup ys = ys \cup xs$. The order is "forgotten" in the sense that it doesn't matter how two multisets are union-ed together, such as, $\wr a, a, b, c \int = \wr b, a, c, a \int$ are equal as finite multisets (justifying the bag notation). This is unlike free monoids, where $[a, a, b, c] \neq [b, a, c, a]$ (justifying the list notation).

5.1 Free monoids with a quotient

Instead of constructing free commutative monoids directly, the first construction we study is to take *any* free monoid and quotient by *symmetries*. Specific instances of this construction are given in §§ 5.2 and 5.4.

From the universal algebraic perspective developed in § 3, this means to extend the equational theory of a given algebraic signature with symmetries. If $(\mathfrak{F}(A), \eta)$ is a free monoid construction satisfying its universal property, then we'd like to quotient $F(A)$ by an *appropriate* symmetry relation $\approx$. This is exactly a *permutation relation*!

Definition 5.1 (Permutation relation). A binary relation $\approx$ on free monoids is a correct permutation relation iff it:

- is reflexive, symmetric, transitive (an equivalence),
- is a congruence wrt $\bullet$: $a \approx b \to c \approx d \to a \bullet c \approx b \bullet d$,
- is commutative: $a \bullet b \approx b \bullet a$, and
- respects $(-)^\sharp$: $\forall f,\ a \approx b \to f^\sharp(a) = f^\sharp(b)$.

Let $q: F(A) \twoheadrightarrow F(A)/\!\!/\!\approx$ be the quotient (inclusion) map. The generators map is given by $q \circ \eta_A$, the identity element is $q(e)$, and the $\bullet$ operation is lifted to the quotient by congruence.

Proposition 8. *$(\mathfrak{F}(A)/\!\!/\!\approx, \bullet, q(e))$ is a commutative monoid.*

For clarity, we will use $\widehat{(-)}$ to denote the extension operation of $F(A)$, and $(-)^\sharp$ for the extension operation of $F(A)/\!\!/\!\approx$.

Definition 5.2. Given a commutative monoid $\mathfrak{X}$ and a map $f: A \to X$, we define $f^\sharp: \mathfrak{F}(A)/\!\!/\!\approx\ \to \mathfrak{X}$ as follows: we first obtain $\widehat{f}: \mathfrak{F}(A) \to \mathfrak{X}$ by universal property of F, and lift it to $\mathfrak{F}(A)/\!\!/\!\approx\ \to \mathfrak{X}$, which is allowed since $\approx$ respects $(-)^\sharp$.

Proposition 9 (Universal property for $\mathfrak{F}(A)/\!\!/\!\approx$)**.** *$(\mathfrak{F}(A)/\!\!/\!\approx, \eta_A: A \xrightarrow{\eta_A} \mathfrak{F}(A) \xrightarrow{q} \mathfrak{F}(A)/\!\!/\!\approx)$ is the free comm. monoid on A.*

5.2 Lists quotiented by permutation

A specific instance of this construction is List quotiented by a permutation relation to get commutativity. We study one such construction (PList), considered in [Joram and Veltri 2023], who give a proof that PList is equivalent to the free commutative monoid (constructed as a HIT). We give a direct proof of its universal property using our generalisation.

Of course, there are many permutation relations in the literature, we consider the one which swaps any two adjacent elements somewhere in the middle of the list.

Definition 5.3 (PList).

```
data Perm (A : 𝓤) : List A → List A → 𝓤 where
  perm-refl : ∀ {xs} → Perm xs xs
  perm-swap : ∀ {x y xs ys zs}
            → Perm (xs ++ x ∷ y ∷ ys) zs
            → Perm (xs ++ y ∷ x ∷ ys) zs

PList : 𝓤 → 𝓤
PList A = List A // Perm
```

By § 5.1, it suffices to show Perm satisfies the axioms of permutation relation to show PList is the free commutative monoid.

Proposition 10. *Let $\mathfrak{X}$ be a commutative monoid, and $f: A \to X$. For $x, y: A$ and xs, ys: $\mathsf{PList}(A)$, $f^\sharp(xs +\!\!+\ x :: y :: ys) = f^\sharp(xs +\!\!+\ y :: x :: ys)$. Hence,* Perm *respects $(-)^\sharp$.*

Remarks. With this representation it is very easy to lift functions and properties defined on List to PList since PList is a quotient of List. The inductive nature of PList makes it very easy to define algorithms and proofs that are inductive in nature, e.g. defining insertion sort on PList is very simple since insertion sort inductively sorts a list, which we can easily do by pattern matching on PList since the construction of PList is definitionally inductive. This property also makes it such that oftentimes inductively constructed PList would normalize to the simplest form of the PList, e.g. $q([x]) ++ q([y, z])$ normalizes to $q([x, y, z])$ by definition, saving the efforts of defining auxillary lemmas to prove their equality.

This inductive nature, however, makes it difficult to define efficient operations on PList. Consider a divide-and-conquer algorithm such as merge sort, which involves partitioning a PList of length $n + m$ into two smaller PList of length n and length m. The inductive nature of PList makes it such that we must iterate all n elements before we can make such a partition, thus making PList unintuitive to work with when we want to reason with operations that involve arbitrary partitioning.

5.3 Swap-List

Informally, quotients are defined by generating all the points, then quotienting out into equivalence classes by the congruence relation. Alternately, HITs use generators (points) and higher generators (paths) (and higher higher generators and so on...). We can define free commutative monoids using HITs were adjacent swaps generate all symmetries, for example swap-lists taken from [Choudhury and Fiore 2023] (and in the Cubical library).

```
data SList (A : 𝓤) : 𝓤 where
  [] : SList A
  _∷_ : A → SList A → SList A
  swap : ∀ x y xs → x ∷ y ∷ xs ≡ y ∷ x ∷ xs
  trunc : ∀ x y → (p q : x ≡ y) → p ≡ q
```

Remarks. Much like PList and List, SList is inductively defined, therefore making it very intuitive to reason with when defining inductive operations or inductive proofs on SList, however difficult to reason with when defining operations that involve arbitrary partitioning, for reasons similar to those given in § 5.2.

5.4 Bag

Alternatively, we can also quotient Array by symmetries to get commutativity. This construction is first considered in [Altenkirch et al. 2011] and [Li 2015], then partially considered in [Choudhury and Fiore 2023], and also in [Joram and Veltri 2023], who gave a similar construction, where only the index function is quotiented, instead of the entire array. [Danielsson 2012] also considered Bag as a setoid relation on List in an intensional MLTT setting. [Joram and Veltri 2023] prove that their version of Bag is the free commutative monoid by equivalence to the other HIT constructions. We give a direct proof of its universal property instead, using the technology we have developed.

Definition 5.4 (Bag).

```
_≈_ : Array A → Array A → 𝓤
(n , f) ≈ (m , g) = Σ(σ : Fin n ≃ Fin m) v = w ∘ σ

Bag : 𝓤 → 𝓤
Bag A = Array A // _≈_
```

Note that by the pigeonhole principle, σ can only be constructed when $n = m$, though this requires a proof in type theory (see the formalization). Conceptually, we are quotienting Array by an automorphism on the indices.

We have already given a proof of Array being the free monoid in § 4.2. By § 5.1 it suffices to show ≈ satisfies the axioms of permutation relations to show that Bag is the free commutative monoid.

Proposition 11. *≈ is a equivalence relation.*

Proposition 12. *≈ is congruent wrt to ++.*

Proof. Given $(n, f) \approx (m, g)$ by σ and $(u, p) \approx (v, q)$ by ϕ, we want to show $(n, f) ++ (u, p) \approx (m, g) ++ (v, q)$ by some τ. We construct τ as follows:

$$\tau := \mathsf{Fin}_{n+u} \xrightarrow{\sim} \mathsf{Fin}_n + \mathsf{Fin}_u \xrightarrow{\sigma,\phi} \mathsf{Fin}_m + \mathsf{Fin}_v \xrightarrow{\sim} \mathsf{Fin}_{m+v}$$

which operationally performs:

$$\begin{tikzcd} \{0, 1, \ldots, n-1, n, n+1, \ldots, n+u-1\} \arrow[d, "\sigma,\phi"] \\ \{\sigma(0), \sigma(1) \ldots, \sigma(n-1), \phi(0), \phi(1), \ldots, \phi(u-1)\} \end{tikzcd}$$

.

□

Proposition 13. *≈ is commutative.*

Proof. We want to show for any arrays (n, f) and (m, g), $(n, f) \bullet (m, g) \approx (m, g) \bullet (n, f)$ by some ϕ. We can use combinators from formal operations in symmetric rig groupoids [Choudhury, Karwowski, et al. 2022] to define ϕ:

$$\phi := \mathsf{Fin}_{n+m} \xrightarrow{\sim} \mathsf{Fin}_n + \mathsf{Fin}_m \xrightarrow{\mathsf{swap}_+} \mathsf{Fin}_m + \mathsf{Fin}_n \xrightarrow{\sim} \mathsf{Fin}_{m+n}$$

which operationally performs:

$$\begin{tikzcd} \{0, 1, \ldots, n-1, n, n+1, \ldots, n+m-1\} \arrow[d, "\phi"] \\ \{n, n+1 \ldots, n+m-1, 0, 1, \ldots, n-1\} \end{tikzcd}$$

□

Proposition 14. *≈ respects $(-)^\sharp$ for arrays.*

It suffices to show that $f^\sharp$ is invariant under permutation: for all $\phi \colon \mathsf{Fin_n} \xrightarrow{\sim} \mathsf{Fin_n}$, $f^\sharp(n, i) = f^\sharp(n, i \circ \phi)$. To prove this we first need to develop some formal combinatorics of *punching in* and *punching out* indices. These operations are borrowed from [Mozler 2021] and developed further in [Choudhury, Karwowski, et al. 2022] for studying permutation codes.

Lemma 4. *Given* $\phi \colon \mathsf{Fin_{S(n)}} \xrightarrow{\sim} \mathsf{Fin_{S(n)}}$*, there is a permutation* $\tau \colon \mathsf{Fin_{S(n)}} \xrightarrow{\sim} \mathsf{Fin_{S(n)}}$ *such that* $\tau(0) = 0$*, and* $f^\sharp(S(n), i \circ \phi) = f^\sharp(S(n), i \circ \tau)$*.*

Proof. Let k be $\phi^{-1}(0)$, and $k + j = S(n)$, we construct τ:

$$\tau \coloneqq \mathsf{Fin_{S(n)}} \xrightarrow{\phi} \mathsf{Fin_{S(n)}} \xrightarrow{\sim} \mathsf{Fin_{k+j}} \xrightarrow{\sim} \mathsf{Fin_k} + \mathsf{Fin_j}$$

$$\xrightarrow{\mathsf{swap}_+} \mathsf{Fin_j} + \mathsf{Fin_k} \xrightarrow{\sim} \mathsf{Fin_{j+k}} \xrightarrow{\sim} \mathsf{Fin_{S(n)}}$$

$$\begin{tikzcd}
\{0, 1, 2, \ldots, k, k+1, k+2, \ldots\} \arrow[d, "\phi"] & \{0, 1, 2, \ldots, k, k+1, k+2, \ldots\} \arrow[d, "\tau"] \\
\{x, y, z, \ldots, 0, u, v, \ldots\} & \{0, u, v, \ldots, x, y, z, \ldots\}
\end{tikzcd}$$

It is trivial to show $f^\sharp(S(n), i \circ \phi) = f^\sharp(S(n), i \circ \tau)$, since the only operation on indices in τ is swap_+. It suffices to show $(S(n), i \circ \phi)$ can be decomposed into two arrays such that $(S(n), i \circ \phi) = (k, g) +\!\!+ (j, h)$ for some g and h. Since the image of $f^\sharp$ is a commutative monoid, and $f^\sharp$ is a homomorphism, $f^\sharp((k, g) +\!\!+ (j, h)) = f^\sharp(k, g) \bullet f^\sharp(j, h) = f^\sharp(j, h) \bullet f^\sharp(k, g) = f^\sharp((j, h) +\!\!+ (k, g))$, thereby proving $f^\sharp(S(n), i \circ \phi) = f^\sharp(S(n), i \circ \tau)$. □

Lemma 5. *Given* $\tau \colon \mathsf{Fin_{S(n)}} \xrightarrow{\sim} \mathsf{Fin_{S(n)}}$ *where* $\tau(0) = 0$*, there is a* $\psi \colon \mathsf{Fin_n} \xrightarrow{\sim} \mathsf{Fin_n}$ *such that* $\tau \circ S = S \circ \psi$*.*

Proof. We construct ψ as $\psi(x) = \tau(S(x)) - 1$. Since τ maps only 0 to 0 by assumption, $\forall x. \tau(S(x)) > 0$, therefore the (-1) is well defined. This is the special case for $k = 0$ in the punch-in and punch-out equivalence for Lehmer codes in [Choudhury, Karwowski, et al. 2022].

$$\begin{tikzcd}
\{0, 1, 2, 3, \ldots\} \arrow[d, "\tau"] & \{0, 1, 2, \ldots\} \arrow[d, "\psi"] \\
\{0, x, y, z \ldots\} & \{x-1, y-1, z-1 \ldots\}
\end{tikzcd}$$

□

Theorem 5.5 (Permutation invariance)**.** *For all* $\phi \colon \mathsf{Fin_n} \xrightarrow{\sim} \mathsf{Fin_n}$*,* $f^\sharp(n, i) = f^\sharp(n, i \circ \phi)$*.*

Proof. By induction on n.

- On $n = 0$, $f^\sharp(0, i) = f^\sharp(0, i \circ \phi) = e$.
- On $n = S(m)$,

$$\begin{aligned}
& f^\sharp(S(m), i \circ \phi) & \\
= \; & f^\sharp(S(m), i \circ \tau) & \text{by Lemma 4} \\
= \; & f(i(\tau(0))) \bullet f^\sharp(m, i \circ \tau \circ S) & \text{by definition of } (\_)^\sharp \\
= \; & f(i(0)) \bullet f^\sharp(m, i \circ \tau \circ S) & \text{by construction of } \tau \\
= \; & f(i(0)) \bullet f^\sharp(m, i \circ S \circ \psi) & \text{by Lemma 5} \\
= \; & f(i(0)) \bullet f^\sharp(m, i \circ S) & \text{induction} \\
= \; & f^\sharp(S(m), i) & \text{by definition of } (\_)^\sharp
\end{aligned}$$

□

Remarks. Unlike PList and SList, Bag and its underlying construction Array are not inductively defined, making it difficult to work with when trying to do induction on them. For example, in the proof Proposition 7, two Lemmas 2 and 3 are needed to do induction on Array, as opposed to List and its quotients, where we can do induction simply by pattern matching. Much like PList, when defining functions on Bag, we need to show they respect $\approx$, i.e. $as \approx bs \to f(as) = f(bs)$. This is notably much more difficult than PList and SList, because whereas with PList and SList we only need to consider "small permutations" (i.e. swapping adjacent elements), with Bag we need to consider all possible permutations. For example, in the proof of Theorem 5.5, we need to first construct a τ which satisfies $\tau(0) = 0$ and prove $f^\sharp(n, i \circ \sigma) = f^\sharp(n, i \circ \tau)$ before we can apply induction.

6 Application: Sorting Functions

We will now put to work the universal properties of our types of (ordered) lists and unordered lists, to define operations on them systematically, which are mathematically sound, and reason about them. First, we explore definitions of various operations on both free monoids and free commutative monoids. By univalence (and the structure identity principle), everything henceforth holds for any presentation of free monoids and free commutative monoids, therefore we avoid picking a specific construction. We use $\mathcal{F}(A)$ to denote the free monoid or free commutative monoid on A, $\mathcal{L}(A)$ to exclusively denote the free monoid construction, and $\mathcal{M}(A)$ to exclusively denote the free commutative monoid construction.

For example length is a common operation defined inductively for List, but usually in proof engineering, properties about length, e.g. $\mathsf{length}(xs +\!\!+ ys) = \mathsf{length}(xs) + \mathsf{length}(ys)$, are proven externally. In our framework of free algebras, where the $(\_)^\sharp$ operation is a correct-by-construction homomorphism, we can define operations like length directly by universal extension, which also gives us a proof that they are homomorphisms for free. Note, the fold operation in functional programming is the homomorphism mapping out into the monoid of endofunctions. A further application of the universal property is to prove two different types are equal, by showing they both satisfy the same universal property (see Proposition 1), which is desirable especially when proving a direct equivalence between the two types turns out to be a difficult exercise in combinatorics.

6.1 Prelude

Any presentation of free monoids or free commutative monoids has a length: $\mathcal{F}(A) \to \mathbb{N}$ function. $\mathbb{N}$ is a monoid with $(0, +)$, and further, the $+$ operation is commutative.

Definition 6.1 (length)**.** The length homomorphism is defined as length $:= (\lambda x.\, 1)^\sharp : \mathcal{F}(A) \to \mathbb{N}$.

Going further, any presentation of free monoids or free commutative monoids has a membership predicate $- \in - : A \to \mathcal{F}(A) \to \mathsf{hProp}$, for any set A. For extension, we use the fact that hProp forms a (commutative) monoid under disjunction: $\vee$ and false: $\bot$.

Definition 6.2 ($\in$)**.** The membership predicate on a set A for each element $x : A$ is $x \in - := よ_A(x)^\sharp : \mathcal{F}(A) \to \mathsf{hProp}$, where we define $よ_A(x) := \lambda y.\, x = y : A \to \mathsf{hProp}$.

よ is formally the Yoneda map under the "types are groupoids" correspondence, where $x : A$ is being sent to its representable in the Hom-groupoid (formed by the identity type), of type hProp. Note that the proofs of (commutative) monoid laws for hProp use equality, which requires the use of univalence (or at least, propositional extensionality). By construction, this membership predicate satisfies its homomorphic properties (the colluquial here/there constructors for de Bruijn indices).

We note that hProp is actually one type level higher than A. To make the type level explicit, A is of type level ℓ, and since hProp_ℓ is the type of all types $X : \mathcal{U}_\ell$ that are mere propositions, hProp_ℓ has type level $\ell + 1$. While we can reduce to the type level of hProp_ℓ to ℓ if we assume Voevodsky's propositional resizing axiom [Voevodsky 2011], we chose not to do so and work within a relative monad framework similar to [Choudhury and Fiore 2023, Section 3]. In the formalization, $(-)^\sharp$ is type level polymorphic to accommodate for this. We explain this further in § 7.

More generally, any presentation of free (commutative) monoids $\mathcal{F}(A)$ also supports the Any and All predicates, which allow us to lift a predicate $A \to \mathsf{hProp}$ (on A), to *any* or *all* elements of $xs : \mathcal{F}(A)$, respectively. In fact, hProp forms a (commutative) monoid in two different ways: $(\bot, \vee)$ and $(\top, \wedge)$ (disjunction and conjunction), which are the two different ways to get Any and All, respectively.

Definition 6.3 (Any and All)**.**

$$\mathsf{Any}(P) := P^\sharp : \mathcal{F}(A) \to (\mathsf{hProp}, \bot, \vee)$$
$$\mathsf{All}(P) := P^\sharp : \mathcal{F}(A) \to (\mathsf{hProp}, \top, \wedge)$$

Remark. Note that Cubical Agda has problems with indexing over HITs, hence it is preferable to program with our universal properties, such as when defining Any and All, because the (indexed) inductive definitions of these predicates get stuck on transp terms.

There is a head function on lists, which is a function that returns the first element of a non-empty list. Formally, this is a monoid homomorphism from $\mathcal{L}(A)$ to $1 + A$.

Definition 6.4 (head)**.** The head homomorphism is defined as head $:= \mathsf{inr}^\sharp : \mathcal{L}(A) \to 1 + A$, where the monoid structure on $1 + A$ has unit $e := \mathsf{inl}(\star) : 1 + A$, and multiplication picks the leftmost element that is defined.

$$\begin{array}{lcccl} \mathsf{inl}(\star) & \oplus & b & := & b \\ \mathsf{inr}(a) & \oplus & b & := & \mathsf{inr}(a) \end{array}$$

Note that the monoid operation $\oplus$ is not commutative, since swapping the input arguments to $\oplus$ would return the leftmost or rightmost element. To make it commutative would require a canonical way to pick between two elements – this leads us to the next section.

6.2 Total orders

First, we recall the axioms of a total order $\leq$ on a set A.

Definition 6.5 (Total order)**.** A total order on a set A is a relation $\leq : A \to A \to \mathsf{hProp}$ that satisfies:

- reflexivity: $x \leq x$,
- transitivity: if $x \leq y$ and $y \leq z$, then $x \leq z$,
- antisymmetry: if $x \leq y$ and $y \leq x$, then $x = y$,
- strong-connectedness: $\forall x, y$, either $x \leq y$ or $y \leq x$.

Note that *either-or* means that this is a (truncated) logical disjunction. In the context of this paper, we want to make a distinction between "decidable total order" and "total order". A *decidable* total order requires the $\leq$ relation to be decidable:

- decidability: $\forall x, y$, we have $x \leq y + \neg(x \leq y)$.

This strengthens the strong-connectedness axiom, where we have either $x \leq y$ or $y \leq x$ merely as a proposition, but decidability allows us to actually compute if $x \leq y$ is true.

Proposition 15. *In a decidable total order, it holds that* $\forall x, y, (x \leq y) + (y \leq x)$. *Further, this makes A discrete, that is* $\forall x, y, (x = y) + (x \neq y)$.

An equivalent way to define a total order is using a binary meet operation (without starting from an ordering relation).

Definition 6.6 (Meet semi-lattice)**.** A meet semi-lattice is a set A with a binary operation $- \sqcap - : A \to A \to A$ that is:

- idempotent: $x \sqcap x = x$,
- associative: $(x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$,
- commutative: $x \sqcap y = y \sqcap x$.

A *strongly-connected* meet semi-lattice further satisfies:

- strong-connectedness: $\forall x, y$, either $x \sqcap y = x$ or $x \sqcap y = y$.

A *total* meet semi-lattice strengthens this to:

- totality: $\forall x, y, (x \sqcap y = x) + (x \sqcap y = y)$.

Proposition 16. *A total order $\leq$ on a set A is equivalent to a strongly-connected meet semi-lattice structure on A. Further, a decidable total order on A induces a total meet semi-lattice structure on A.*

Proof sketch. Given a (mere) total order $\leq$ on a set A, we define $x \sqcap y :\equiv \text{if } x \leq y \text{ then } x \text{ else } y$. Crucially, this map is *locally-constant*, allowing us to eliminate from an hProp to an hSet. Meets satisfy the universal property of products, that is, $c \leq a \sqcap b \iff c \leq a \wedge c \leq b$, and the axioms follow by calculation using $\lessgtr$-arguments. Conversely, given a meet semi-lattice, we define $x \leq y :\equiv x \sqcap y = x$, which defines an hProp-valued total ordering relation. If the total order is decidable, we use the discreteness of A from Proposition 15. ☐

Finally, tying this back to Definition 6.4, we have the following for free commutative monoids.

Definition 6.7 (head)**.** Assume a total order $\leq$ on a set A. We define a commutative monoid structure on $1 + A$, with unit $e :\equiv \mathsf{inl}(\star) : 1 + A$, and multiplication defined as:

$$\begin{array}{llll} \mathsf{inl}(\star) & \oplus & b & :\equiv & b \\ \mathsf{inr}(a) & \oplus & \mathsf{inl}(\star) & :\equiv & \mathsf{inr}(a) \\ \mathsf{inr}(a) & \oplus & \mathsf{inr}(b) & :\equiv & \mathsf{inr}(a \sqcap b) \ . \end{array}$$

This gives a homomorphism $\mathsf{head} :\equiv \mathsf{inr}^\sharp : \mathcal{M}(A) \to 1 + A$, which picks out the *least* element of the free commutative monoid.

6.3 Sorting functions

The free commutative monoid is also a monoid, hence, there is a canonical monoid homomorphism $q : \mathcal{L}(A) \to \mathcal{M}(A)$, which is given by ${\eta_A}^\sharp$. Since $\mathcal{M}(A)$ is (upto equivalence), a quotient of $\mathcal{L}(A)$ by symmetries (or a permutation relation), it is a surjection (in particular, a regular epimorphism in Set as constructed in type theory). Concretely, q simply includes the elements of $\mathcal{L}(A)$ into equivalence classes of lists in $\mathcal{M}(A)$, which "forgets" the order that was imposed by the indexing of the list.

Classically, assuming the Axiom of Choice would allow us to construct a section (right-inverse) to the surjection q, that is, a function $s : \mathcal{M}(A) \to \mathcal{L}(A)$ such that $\forall x.\, q(s(x)) = x$. Or in informal terms, given the surjective inclusion into the quotient, a section (uniformly) picks out a canonical representative for each equivalence class. Constructively, does q have a section? If symmetry kills the order, can it be resurrected?

$$\begin{tikzcd} \mathcal{L}(A) \arrow[r, two heads, "q"] & \mathcal{M}(A) \arrow[l, bend left, "s"] \end{tikzcd}$$

Figure 1. Relationship of $\mathcal{L}(A)$ and $\mathcal{M}(A)$

Viewing the quotienting relation as a permutation relation (from § 5.1), a section s to q has to pick out canonical representatives of equivalence classes generated by permutations. Using SList as an example, $s(x :: y :: xs) = s(y :: x :: xs)$ for any $x, y : A$ and $xs : \mathsf{SList}(A)$, and since it must also respect $\forall xs.\, q(s(xs)) = xs$, s must preserve all the elements of xs. It cannot be a trivial function such as $\lambda xs.[]$ – it must produce a permutation of the elements of s! But to place these elements side-by-side in the list, s must somehow impose an order on A (invariant under permutation), turning unordered lists of A into ordered lists of A. Axiom of Choice (AC) giving us a section s to q "for free" is analagous to how AC implies the well-ordering principle, which states every set can be well-ordered. If we assumed AC our problem would be trivial! Instead we study how to constructively define such a section, and in fact, that is exactly the extensional view of a sorting algorithm, and the implications of its existence is that A can be ordered, or sorted.

6.3.1 Section from Order.

Proposition 17. *Assume a decidable total order on A. There is a sort function $s : \mathcal{M}(A) \to \mathcal{L}(A)$ which constructs a section to $q : \mathcal{L}(A) \twoheadrightarrow \mathcal{M}(A)$*

Proof sketch. We can construct such a sort function by implementing any sorting algorithm. In our formalization we chose insertion sort, because it can be defined easily using the inductive structure of $\mathsf{SList}(A)$ and $\mathsf{List}(A)$. To implement other sorting algorithms like mergesort, other representations such as Bag and Array would be preferable, as explained in § 5.4. To see how this proposition holds: $q(s(xs))$ orders an unordered list xs by s, and discards the order again by q – imposing and then forgetting an order on xs simply *permutes* its elements, which proves $q \circ s \sim \mathsf{id}$. ☐

This is not surprising…we want to go the other way.

6.3.2 Order from Section.

The previous section allowed us to construct a section – how do we know this is a *correct* sort function? At this point we ask: if we can construct a section from order, can we construct an order from section? Indeed, just by the virtue of s being a section, we can (almost) construct a total-ordering relation on the carrier set!

Definition 6.8. Given a section s, we define:

$$\begin{aligned} \mathsf{least}(xs) &:\equiv \mathsf{head}(s(xs)) \\ x \preccurlyeq_s y &:\equiv \mathsf{least}(\wr x, y \int) = \mathsf{inr}(x) \ . \end{aligned}$$

That is, we take the two-element bag $\wr x, y \int$, "sort" it by s, and test if the head element is x. Note, this is equivalent to $x \preccurlyeq_s y :\equiv s\wr x, y \int = [x, y]$, because s preserves length, and the second element is forced to be y.

Proposition 18. *$\preccurlyeq_s$ is reflexive, antisymmetric, and total.*

Proof. For all x, $\mathsf{least}(\wr x, x \int)$ must be $\mathsf{inr}(x)$, therefore $x \preccurlyeq_s x$, giving reflexivity. For all x and y, given $x \preccurlyeq_s y$ and $y \preccurlyeq_s x$, we have $\mathsf{least}(\wr x, y \int) = \mathsf{inr}(x)$ and $\mathsf{least}(\wr y, x \int) = \mathsf{inr}(y)$. Since $\wr x, y \int = \wr y, x \int$, $\mathsf{least}(\wr x, y \int) = \mathsf{least}(\wr y, x \int)$, therefore we have $x = y$, giving antisymmetry. For all x and y, $\mathsf{least}(\wr x, y \int)$ is merely either $\mathsf{inr}(x)$ or $\mathsf{inr}(y)$, therefore we have merely either $x \preccurlyeq_s y$ or $y \preccurlyeq_s x$, giving totality. □

Although s correctly orders 2-element bags, it doesn't necessarily sort 3 or more elements – $\preccurlyeq_s$ is not necessarily transitive (a counterexample is given in Proposition 28). We will enforce this by imposing additional constraints on the *image* of s.

Definition 6.9 ($- \in \mathsf{im}(s)$). The fiber of s over $xs : \mathcal{L}(A)$ is given by $\mathsf{fib}_s(xs) :\equiv \sum_{(ys : \mathcal{M}(A))} s(ys) = xs$. The image of s is given by $\mathsf{im}(s) :\equiv \sum_{(xs : \mathcal{L}(A))} \|\mathsf{fib}_s(xs)\|_{-1}$. Simplifying, we say that $xs : \mathcal{L}(A)$ is "in the image of s", or, $xs \in \mathsf{im}(s)$, if there merely exists a $ys : \mathcal{M}(A)$ such that $s(ys) = xs$.

If s *were* a sort function, the image of s would be the set of s-"sorted" lists, therefore $xs \in \mathsf{im}(s)$ would imply xs is a correctly s-"sorted" list. First, we note that the 2-element case is correct.

Proposition 19. $x \preccurlyeq_s y$ *iff* $[x, y] \in \mathsf{im}(s)$.

Then, we state the first axiom on s.

Definition 6.10 (im-cut). A section s satisfies im-cut iff for all x, y, xs:

$$y \in x :: xs \ \wedge\ x :: xs \in \mathsf{im}(s) \ \longrightarrow\ [x, y] \in \mathsf{im}(s) \ .$$

We use the definition of list membership from Definition 6.2. The $\in$ symbol is intentionally overloaded to make the axiom look like a logical "cut" rule. Inforamlly, it says that the head of an s-"sorted" list is the least element of the list.

Proposition 20. *If A has a total order $\leq$, insertion sort defined using $\leq$ satisfies* im-cut.

Proposition 21. *If s satisfies* im-cut, *$\preccurlyeq_s$ is transitive.*

Proof. Given $x \preccurlyeq_s y$ and $y \preccurlyeq_s z$, we want to show $x \preccurlyeq_s z$. Consider the 3-element bag $\wr x, y, z \int : \mathcal{M}(A)$. Let u be $\mathsf{least}(\wr x, y, z \int)$, by Definition 6.10 and Proposition 19, we have $u \preccurlyeq_s x \wedge u \preccurlyeq_s y \wedge u \preccurlyeq_s z$. Since $u \in \wr x, y, z \int$, u must be one of the elements. If $u = x$ we have $x \preccurlyeq_s z$. If $u = y$ we have $y \preccurlyeq_s x$, and since $x \preccurlyeq_s y$ and $y \preccurlyeq_s z$ by assumption, we have $x = y$ by antisymmetry, and then we have $x \preccurlyeq_s z$ by substitution. Finally, if $u = z$, we have $z \preccurlyeq_s y$, and since $y \preccurlyeq_s z$ and $x \preccurlyeq_s y$ by assumption, we have $z = y$ by antisymmetry, and then we have $x \preccurlyeq_s z$ by substitution. □

6.3.3 Embedding orders into sections.

Following from Propositions 18 and 21, and Proposition 20, we have shown that a section s that satisfies im-cut produces a total order $x \preccurlyeq_s y :\equiv \mathsf{least}(\wr x, y \int) = \mathsf{inr}(x)$, and a total order $\leq$ on the carrier set produces a section satisfying im-cut, constructed by sorting with $\leq$. This constitutes an embedding of decidable total orders into sections satisfying im-cut.

Proposition 22. *Assume A has a decidable total order $\leq$, we can construct a section s that satisfies* im-cut, *such that $\preccurlyeq_s$ constructed from s is equivalent to $\leq$.*

Proof. By the insertion sort algorithm parameterized by $\leq$, it holds that $[x, y] \in \mathsf{im}(s)$ iff $x \leq y$. By Proposition 19, we have $x \preccurlyeq_s y$ iff $x \leq y$. We now have a total order $x \preccurlyeq_s y$ equivalent to $x \leq y$. □

6.3.4 Equivalence of order and sections.

We want to upgrade the embedding to an isomorphism, and it remains to show that we can turn a section satisfying im-cut to a total order $\preccurlyeq_s$, then construct the *same* section back from $\preccurlyeq_s$. Unfortunately, this fails (see Proposition 29)! We then introduce our second axiom of sorting.

Definition 6.11 (im-cons). A section s satisfies im-cons iff for all x, xs,

$$x :: xs \in \mathsf{im}(s) \longrightarrow xs \in \mathsf{im}(s)$$

This says that s-"sorted" lists are downwards-closed under cons-ing, that is, the tail of an s-"sorted" list is also s-"sorted". To prove the correctness of our axioms, first we need to show that a section s satisfying im-cut and im-cons is equal to insertion sort parameterized by the $\preccurlyeq_s$ constructed from s. In fact, the axioms we have introduced are equivalent to the standard inductive characterization of sorted lists, found in textbooks, such as in [Appel 2023].

```
data Sorted (≤ : A → A → 𝓤) : List A → 𝓤 where
  sorted-[] : Sorted []
  sorted-η : ∀ x → Sorted [ x ]
  sorted-:: : ∀ x y zs → x ≤ y
     → Sorted (y :: zs) → Sorted (x :: y :: zs)
```

Note that $\mathsf{Sorted}_{\leq}(xs)$ is a proposition for every xs, and forces the list xs to be permuted in a unique way.

Lemma 6. *Given an order $\leq$, for any $xs, ys : \mathcal{L}(A)$, $q(xs) = q(ys) \wedge \mathsf{Sorted}_{\leq}(xs) \wedge \mathsf{Sorted}_{\leq}(ys) \longrightarrow xs = ys$.*

Insertion sort by $\leq$ always produces lists that satisfy $\mathsf{Sorted}_{\leq}$. Functions that also produce lists satisfying $\mathsf{Sorted}_{\leq}$ are equal to insertion sort by function extensionality.

Proposition 23. *Given an order $\leq$, if a section s always produces sorted list, i.e. $\forall xs.\ \mathsf{Sorted}_{\leq}(s(xs))$, s is equal to insertion sort by $\leq$.*

Finally, this gives us correctness of our axioms.

Proposition 24. *Given a section s that satisfies* im-cut *and* im-cons, *and $\preccurlyeq_s$ the order derived from s, then for all $xs : \mathcal{M}(A)$, it holds that $\mathsf{Sorted}_{\preccurlyeq_s}(s(xs))$. Equivalently, for all lists $xs : \mathcal{L}(A)$, it holds that $xs \in \mathsf{im}(s)$ iff $\mathsf{Sorted}_{\preccurlyeq_s}(xs)$.*

Proof. We inspect the length of $xs : \mathcal{M}(A)$. For lengths 0 and 1, this holds trivially. Otherwise, we proceed by induction: given a $xs : \mathcal{M}(A)$ of length ≥ 2, let $s(xs) = x :: y :: ys$. We need to show $x \preccurlyeq_s y \wedge \mathsf{Sorted}_{\preccurlyeq_s}(y :: ys)$ to construct $\mathsf{Sorted}_{\preccurlyeq_s}(x :: y :: ys)$. By im-cut, we have $x \preccurlyeq_s y$, and by im-cons, we inductively prove $\mathsf{Sorted}_{\preccurlyeq_s}(y :: ys)$. □

Lemma 7. *Given a decidable total order $\leq$ on A, we can construct a section $t_{\leq}$ satisfying* im-cut *and* im-cons, *such that, for the order $\preccurlyeq_s$ derived from s, we have $t_{\preccurlyeq_s} = s$.*

Proof. From s we can construct a decidable total order $\preccurlyeq_s$ since s satisfies im-cut and A has decidable equality by assumption. We construct $t_{\preccurlyeq_s}$ as insertion sort parameterized by $\preccurlyeq_s$ constructed from s. By Proposition 23 and Proposition 24, $s = t_{\preccurlyeq_s}$. □

Proposition 25. *Assume A has a decidable total order $\leq$, then A has decidable equality.*

Proof. We decide if $x \leq y$ and $y \leq x$, and by cases:

- if $x \leq y$ and $y \leq x$: by antisymmetry, $x = y$.
- if $\neg(x \leq y)$ and $y \leq x$: assuming $x = y$, have $x \leq y$, leading to contradiction by $\neg(x \leq y)$, hence $x \neq y$.
- if $x \leq y$ and $\neg(y \leq x)$: similar to the previous case.
- if $\neg(x \leq y)$ and $\neg(y \leq x)$: by totality, either $x \leq y$ or $y \leq x$, which leads to a contradiction.

□

We can now state and prove our main theorem.

Definition 6.12 (Sorting function)**.** A sorting function is a section $s : \mathcal{M}(A) \to \mathcal{L}(A)$ to the canonical surjection $q : \mathcal{L}(A) \twoheadrightarrow \mathcal{M}(A)$ satisfying two axioms:

- im-cut: $x :: xs \in \mathsf{im}(s) \wedge y \in x :: xs \to [x, y] \in \mathsf{im}(s)$,
- im-cons: $x :: xs \in \mathsf{im}(s) \to xs \in \mathsf{im}(s)$.

Theorem 6.13. *Let* $\mathsf{DecTotOrd}(A)$ *be the set of decidable total orders on A,* $\mathsf{Sort}(A)$ *be the set of correct sorting functions with carrier set A, and* $\mathsf{Discrete}(A)$ *be a predicate which states A has decidable equality. There is a map* $o2s : \mathsf{DecTotOrd}(A) \to \mathsf{Sort}(A) \times \mathsf{Discrete}(A)$, *which is an equivalence.*

Proof. $o2s$ is constructed by parameterizing insertion sort with $\leq$. By Proposition 25, A is Discrete. The inverse $s2o(s)$ is constructed by Definition 6.8, which produces a total order by Propositions 18 and 21, and a decidable total order by $\mathsf{Discrete}(A)$. By Proposition 22 we have $s2o \circ o2s = \mathsf{id}$, and by Lemma 7 we have $o2s \circ s2o = \mathsf{id}$, giving an isomorphism, hence an equivalence. □

Remarks. The sorting axioms we have come up with are abstract properties of functions. As a sanity check, we can verify that the colloquial correctness specification of a sorting function (starting from a total order) matches our axioms. We consider the correctness criterion developed in [Alexandru 2023].

Proposition 26. *Assume a decidable total order $\leq$ on A. A sorting algorithm is a map* $\mathsf{sort} : \mathcal{L}(A) \to \mathcal{OL}(A)$, *that turns lists into ordered lists, where $\mathcal{OL}(A)$ is defined as* $\sum_{(xs : \mathcal{L}(A))} \mathsf{Sorted}_{\leq}(xs)$, *such that:*

$$\begin{tikzcd} \mathcal{L}(A) \arrow[rr, "\mathsf{sort}"] \arrow[dr, "q"'] & & \mathcal{OL}(A) \arrow[dl, "q \circ \pi_1"] \\ & \mathcal{M}(A) & \end{tikzcd}$$

Sorting functions give sorting algorithms.

Proof. We construct our section $s : \mathcal{M}(A) \to \mathcal{L}(A)$, and define $\mathsf{sort} := s \circ q$, which produces ordered lists by Proposition 24. □

7 Formalization

In this section, we discuss some aspects of the formalization. The paper uses informal type theoretic language, and is accessible without understanding any details of the formalization. However, the formalization is done in Cubical Agda, which has a few differences and a few shortcomings due to proof engineering issues.

For simplicity we omitted type levels in the paper, but our formalization has many verbose uses of universe levels due to Agda's universe polymorphism. Similarly, h-levels were restricted to sets in the paper, but the formalization is parameterized in many places for any h-level (to facilitate future generalizations). The free algebra framework currently only works with sets. Due to issues of regularlity, certain computations only hold propositionally, and the formalization requires proving auxiliary β and η computation rules in somce places. We also note the axioms of sorting in the formalization are named differently from the paper. We give a table of the Agda module names and their corresponding sections in the paper in Table 1.

8 Discussion

We conclude by discussing some high-level observations, related work, and future directions.

Free commutative monoids. The construction of finite multisets and free commutative monoids has a long history, and various authors have different approaches to it. We refer the reader to the discussions in [Choudhury and Fiore 2023; Joram and Veltri 2023] for a detailed survey of these constructions. Our work, in particular, was motivated by the colloquial observation that: "there is no way to represent free commutative monoids using inductive types". From the categorical point of view, this is simply the fact that the free commutative monoid endofunctor on Set is not polynomial (doesn't preserve pullbacks). This has led various authors to think about clever encodings of free commutative monoids using inductive types by adding assumptions on the carrier set – in particular, the assumption of total ordering on

Module	Description	Reference
`index`	Index of all files	N/A
`Cubical.Structures.Free`	Free algebras	§ 3.2
`Cubical.Structures.Sig`	Algebraic signatures	Definition 3.1
`Cubical.Structures.Str`	Algebraic structures	Definition 3.5
`Cubical.Structures.Eq`	Equational theories	§ 3.3
`Cubical.Structures.Tree`	Trees	Definition 3.11
`Cubical.Structures.Set.Mon.List`	Lists	§ 4.1
`Cubical.Structures.Set.Mon.Array`	Arrays	§ 4.2
`Cubical.Structures.Set.CMon.QFreeMon`	Quotiented-free monoid	§ 5.1
`Cubical.Structures.Set.CMon.PList`	Quotiented-list	§ 5.2
`Cubical.Structures.Set.CMon.SList`	Swapped-list	§ 5.3
`Cubical.Structures.Set.CMon.Bag`	Bag	§ 5.4
`Cubical.Structures.Set.CMon.SList.Sort`	Sort functions	§ 6

Table 1. Status of formalised results

the carrier set leads to the construction of "fresh-lists", by [Kupke et al. 2023], which forces the canonical *sorted* ordering on the elements of the finite multiset.

It is worth noting that in programming practice, it is usually the case that all user-defined types have some sort of total order enforced on them, either because they're finite, or they can be enumerated in some way. Therefore, under these assumptions, the construction of fresh lists is a very reasonable way to represent free commutative monoids, or finite multisets.

Correctness of Sorting. Sorting is a classic problem in computer science, and the functional programming view of sorting and its correctness has been studied by various authors. The simplest view of sorting is a function sort : $\mathcal{L}(\mathbb{N}) \to \mathcal{L}(\mathbb{N})$, which permutes the list and outputs an ordered list, which is studied in [Appel 2023]. Fundamentally, this is a very extrinsic view of program verification, which is common in the *Coq* community, and further, a very special case of a more general sorting algorithm.

Henglein in "What Is a Sorting Function?" [Henglein 2009], studies sorting functions abstractly, without requiring a total order on the underlying set. He considers sorting functions as functions on sequences (lists), and recovers the order by "sorting" an n-element list, and looking up the position of the elements to be compared. Unlike us, Henglein does not factorize the sorting function through free commutative monoids, but the ideas are extremely similar. We are able to give a more refined axiomatization of sorting because we consider the symmetries, or permutations, explicitly, and work in a constructive setting (using explicit assumptions about decidability), and this is a key improvement over this previous work.

The other more refined intrinsic view of correct sorting has been studied in [Hinze et al. 2012], and further expanded in [Alexandru 2023], which matches our point of view, as explained in Proposition 26. However, their work is not just about extensional correctness of sorting, but also deriving various sorting algorithms starting from bialgebraic semantics and distributive laws. Our work is complementary to theirs, in that we are not concerned with the computational content of sorting, but rather the abstract properties of sorting functions, which are independent of a given ordering. It remains to be seen how these ideas could be combined – the abstract property of sorting, with the intrinsic essence of sorting algortihms – and that is a direction for future work. This paper only talks about sorting lists and bags, but the abstract property of correct sorting functions could be applied to more general inductive types. We speculate that this could lead to some interesting connections with sorting (binary) trees, and constructions of (binary) search trees, from classical computer science.

Universal Algebra. One of the contributions of our work is also a rudimentary framework for universal algebra, but done in a more categorical style, which lends itself to an elegant formalization in type theory. We believe this framework could be improved and generalised to higher dimensions, moving from sets to groupoids, and using a system of coherences on top of a system of equations, which we are already pursuing. Groupoidyfing free (commutative) monoids to free (symmetric) monoidal groupoids is a natural next step, and its connections to assumptions about total orders on the type of objects would be an important direction to explore.

References

Garrett Birkhoff. Oct. 1935. "On the Structure of Abstract Algebras." *Mathematical Proceedings of the Cambridge Philosophical Society*, 31, 4, (Oct. 1935), 433–454. DOI: 10.1017/S0305004100013463.

Eduardo J Dubuc. May 1, 1974. "Free Monoids." *Journal of Algebra*, 29, 2, (May 1, 1974), 208–228. DOI: 10.1016/0021-8693(74)90095-7.

G. M. Kelly. Aug. 1980. "A Unified Treatment of Transfinite Constructions for Free Algebras, Free Monoids, Colimits, Associated Sheaves, and so

On." *Bulletin of the Australian Mathematical Society*, 22, 1, (Aug. 1980), 1–83. DOI: 10.1017/S0004972700006353.

Andreas Blass. 1983. "Words, Free Algebras, and Coequalizers." *Fundamenta Mathematicae*, 117, 2, 117–160. DOI: 10.4064/fm-117-2-117-160.

Edsger Wybe Dijkstra. Sept. 1997. *A Discipline of Programming*. (1st ed.). Prentice Hall PTR, USA, (Sept. 1997). 240 pp. ISBN: 978-0-13-215871-8.

Michael Abbott, Thorsten Altenkirch, and Neil Ghani. 2003. "Categories of Containers." In: *Foundations of Software Science and Computation Structures* (Lecture Notes in Computer Science). Ed. by Andrew D. Gordon. Springer, Berlin, Heidelberg, 23–38. ISBN: 978-3-540-36576-1. DOI: 10.1007/3-540-36576-1_2.

Fritz Henglein. Aug. 2009. "What Is a Sorting Function?" *The Journal of Logic and Algebraic Programming*, 78, 7, (Aug. 2009), 552–572. DOI: 10.1016/j.jlap.2008.12.003.

Thorsten Altenkirch, Thomas Anberrée, and Nuo Li. 2011. "Definable Quotients in Type Theory." (2011). http://www.cs.nott.ac.uk/~psztxa/publ/defquotients.pdf.

Vladimir Voevodsky. 2011. "Resizing Rules - Their Use and Semantic Justification." TYPES (Bergen). (2011). https://www.math.ias.edu/vladimir/sites/math.ias.edu.vladimir/files/2011_Bergen.pdf.

Nils Anders Danielsson. 2012. "Bag Equivalence via a Proof-Relevant Membership Relation." In: *Interactive Theorem Proving* (Lecture Notes in Computer Science). Ed. by Lennart Beringer and Amy Felty. Springer, Berlin, Heidelberg, 149–165. ISBN: 978-3-642-32347-8. DOI: 10.1007/978-3-642-32347-8_11.

Ralf Hinze, Daniel W.H. James, Thomas Harper, Nicolas Wu, and José Pedro Magalhães. Sept. 12, 2012. "Sorting with Bialgebras and Distributive Laws." In: *Proceedings of the 8th ACM SIGPLAN Workshop on Generic Programming* (WGP '12). Association for Computing Machinery, New York, NY, USA, (Sept. 12, 2012), 69–80. ISBN: 978-1-4503-1576-0. DOI: 10.1145/2364394.2364405.

The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Univalent Foundations Program, Institute for Advanced Study. https://homotopytypetheory.org/book.

Nuo Li. July 15, 2015. *Quotient Types in Type Theory*. (July 15, 2015). Retrieved Apr. 29, 2022 from http://eprints.nottingham.ac.uk/28941/.

Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. July 26, 2019. "Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types." *Proceedings of the ACM on Programming Languages*, 3, (July 26, 2019), 87:1–87:29, ICFP, (July 26, 2019). DOI: 10.1145/3341691.

[SW] Martin Mozler, *Cubical Agda: Simple Application of Fin: Lehmer Codes* 2021. Agda Github Community. URL: https://github.%20com/agda/cubical/blob/a1d2bb38c0794f3cb00610cd6061cf9b5410518d/Cubical/Data/Fin/LehmerCode.agda.

Vikraman Choudhury, Jacek Karwowski, and Amr Sabry. Jan. 11, 2022. "Symmetries in Reversible Programming: From Symmetric Rig Groupoids to Reversible Programming Languages." *Proceedings of the ACM on Programming Languages*, 6, (Jan. 11, 2022), 6:1–6:32, POPL, (Jan. 11, 2022). DOI: 10.1145/3498667.

G. C. (Cass) Alexandru. 2023. "Intrinsically Correct Sorting Using Bialgebraic Semantics." Master's thesis. Radboud University. https://www.ru.nl/icis/education/master-thesis/vm/theses-archive/.

Andrew W. Appel. Aug. 23, 2023. *Verified Functional Algorithms*. Software Foundations. Vol. 3. (Aug. 23, 2023). https://softwarefoundations.cis.upenn.edu/vfa-current/index.html.

Vikraman Choudhury and Marcelo Fiore. Feb. 22, 2023. "Free Commutative Monoids in Homotopy Type Theory." *Electronic Notes in Theoretical Informatics and Computer Science*, Volume 1 - Proceedings of... (Feb. 22, 2023), 10492. DOI: 10.46298/entics.10492.

Philipp Joram and Niccolò Veltri. 2023. "Constructive Final Semantics of Finite Bags." In: *DROPS-IDN/v2/Document/10.4230/LIPIcs.ITP.2023.20*. 14th International Conference on Interactive Theorem Proving (ITP 2023). Schloss-Dagstuhl - Leibniz Zentrum für Informatik. DOI: 10.4230/LIPIcs.ITP.2023.20.

Clemens Kupke, Fredrik Nordvall Forsberg, and Sean Watters. 2023. "A Fresh Look at Commutativity: Free Algebraic Structures via Fresh Lists." In: *Programming Languages and Systems* (Lecture Notes in Computer Science). Ed. by Chung-Kil Hur. Springer Nature, Singapore, 135–154. ISBN: 978-981-9983-11-7. DOI: 10.1007/978-981-99-8311-7_7.

A Supplementary material for Section 2 (Notation)

A.1 Function extensionality

Within the scope of our work, funExt is heavily used in § 4.2 and § 5.4, where a n-element array A^n is defined as lookup functions $\mathsf{Fin_n} \longrightarrow A$. Therefore, to prove two arrays are equal, we need to show that two functions would be equal, which is impossible to do without funExt.

A.2 Higher Inductive Types

In our work, higher inductive types and set quotients are used extensively to define commutative data structures, which we would demonstrate in § 5.

A.3 Univalence

Within the scope of our work, we want to primarily work with sets, therefore we add the truncation constructor whenever necessary so we need not concern ourselves with higher-dimensional paths (or equalities). Since we have multiple constructions of free monoids and free commutative monoids, given in § 4 and § 5, having univalence allows us to easily transport proofs and functions from one construction to another. Another instance where univalence is used is the definition of membership proofs in **??**, where we want to show to propositions are commutative: i.e. $\forall p, q \colon \mathsf{hProp}, p \vee q = q \vee p$. Since p and q are types, we need univalence to show $p \vee q = q \vee p$ are in fact equal.

B Supplementary material for Section 3 (Universal Algebra)

Proposition 1. *Suppose $\mathfrak{F}(X)$ and $\mathfrak{G}(X)$ are both free σ-algebras on X. Then $\mathfrak{F}(X) \simeq \mathfrak{G}(X)$, natural in X.*

Proof. By extending η_X for each free construction, we have maps in each direction: $G\cdot\eta_X{}^\sharp \colon \mathfrak{F}(X) \longrightarrow \mathfrak{G}(X)$, and vice versa. Finally, using Definition 3.10, we have $F\cdot\eta_X{}^\sharp \circ G\cdot\eta_X{}^\sharp \sim (F\cdot\eta_X{}^\sharp \circ G\cdot\eta_X)^\sharp \sim F\cdot\eta_X{}^\sharp \sim \mathsf{id}_{\mathfrak{F}(X)}$. ◻

The free algebra construction automatically turns F into an endofunctor on Set, where the action on functions is given by: $X \xrightarrow{f} Y \mapsto F(X) \xrightarrow{(\eta_Y \circ f)^\sharp} F(Y)$. Further, this gives a monad on Set, with unit given by η, and multiplication given by $\mu_X \coloneqq F(F(X)) \xrightarrow{\mathsf{id}_{F(X)}{}^\sharp} F(X)$.

The free algebra on the empty set $\mathfrak{F}(\mathbf{0})$ is inhabited by all the constant symbols in the signature. We note a few important properties of free algebras on 0, 1, and coproducts.

Proposition 27.

- *σ-$\mathsf{Alg}(\mathfrak{F}(\mathbf{0}), \mathfrak{X})$ is contractible,*
- *if σ has one constant symbol, then $\mathfrak{F}(\mathbf{0})$ is contractible,*
- *the type of algebra structures on $\mathbf{1}$ is contractible,*
- *$\mathfrak{F}(X + Y)$ is the coproduct of $\mathfrak{F}(X)$ and $\mathfrak{F}(Y)$ in σ-Alg:*

$$\sigma\text{-}\mathsf{Alg}(\mathfrak{F}(X+Y), \mathfrak{Z}) \simeq \sigma\text{-}\mathsf{Alg}(\mathfrak{F}(X), \mathfrak{Z}) \times \sigma\text{-}\mathsf{Alg}(\mathfrak{F}(Y), \mathfrak{Z}) \ .$$

Proof. F being a left adjoint, preserves coproducts. This makes $\mathfrak{F}(\mathbf{0})$ initial in σ-Alg. $\mathfrak{F}(\mathbf{1}) \longrightarrow \mathbf{1}$ is contractible because $\mathbf{1}$ is terminal in Set. ◻

C Supplementary material for Section 4 (Constructions of Free Monoids)

Definition C.1 (Concatenation)**.** We define the concatenation operation $+\!\!+ \colon \mathsf{List}(A) \longrightarrow \mathsf{List}(A) \longrightarrow \mathsf{List}(A)$, by recursion on the first argument:

$$[] +\!\!+ ys = ys$$
$$(x :: xs) +\!\!+ ys = x :: (xs +\!\!+ ys)$$

The proof that $+\!\!+$ satisfies monoid laws is straightforward (see the formalization for details).

Definition C.2 (Universal extension)**.** For any monoid $\mathfrak{X}$, and given a map $f \colon A \longrightarrow X$, we define the extension $f^\sharp \colon \mathsf{List}(A) \longrightarrow \mathfrak{X}$ by recursion on the list:

$$f^\sharp([]) = e$$
$$f^\sharp(x :: xs) = f(x) \bullet f^\sharp(xs)$$

Proposition 3. *$(-)^\sharp$ lifts a function $f : A \to X$ to a monoid homomorphism $f^\sharp : \mathsf{List}(A) \to \mathfrak{X}$.*

Proof. To show that $f^\sharp$ is a monoid homomorphism, we need to show $f^\sharp(xs \mathbin{+\!\!+} ys) = f^\sharp(xs) \bullet f^\sharp(ys)$. We can do so by induction on xs.

Case []: $f^\sharp([] \mathbin{+\!\!+} ys) = f^\sharp(ys)$, and $f^\sharp([]) \bullet f^\sharp(ys) = e \bullet f^\sharp(ys) = f^\sharp(ys)$ by definition of $(-)^\sharp$. Therefore, we have $f^\sharp([] \mathbin{+\!\!+} ys) = f^\sharp([]) \bullet f^\sharp(ys)$.

Case $x :: xs$:

$$\begin{aligned}
& f^\sharp((x :: xs) \mathbin{+\!\!+} ys) & \\
&= f^\sharp(([x] \mathbin{+\!\!+} xs) \mathbin{+\!\!+} ys) & \text{by definition of concatenation} \\
&= f^\sharp([x] \mathbin{+\!\!+} (xs \mathbin{+\!\!+} ys)) & \text{by associativity} \\
&= f^\sharp(x :: (xs \mathbin{+\!\!+} ys)) & \text{by definition of concatenation} \\
&= f(x) \bullet f^\sharp(xs \mathbin{+\!\!+} ys) & \text{by definition of } (-)^\sharp \\
&= f(x) \bullet (f^\sharp(xs) \bullet f^\sharp(ys)) & \text{by induction} \\
&= (f(x) \bullet f^\sharp(xs)) \bullet f^\sharp(ys) & \text{by associativity} \\
&= f^\sharp(x :: xs) \bullet f^\sharp(ys) & \text{by definition of } (-)^\sharp
\end{aligned}$$

Therefore, $(-)^\sharp$ does correctly lift a function to a monoid homomorphism. □

Proposition 4 (Universal property for List)**.** *$(\mathsf{List}(A), \eta_A)$ is the free monoid on A.*

Proof. To show that $(-)^\sharp$ is an inverse to $- \circ \eta_A$, we first show $(-)^\sharp$ is the right inverse to $- \circ \eta_A$. For all f and x, $(f^\sharp \circ \eta_A)(x) = f^\sharp(x :: []) = f(x) \bullet e = f(x)$, therefore by function extensionality, for any f, $f^\sharp \circ \eta_A = f$, and $(- \circ \eta_A) \circ (-)^\sharp = id$.

To show $(-)^\sharp$ is the left inverse to $- \circ \eta_A$, we need to prove for any monoid homomorphism $f : \mathsf{List}(A) \to \mathfrak{X}$, $(f \circ \eta_A)^\sharp(xs) = f(xs)$. We can do so by induction on xs.

Case []: $(f \circ \eta_A)^\sharp([]) = e$ by definition of the $(-)^\sharp$ operation, and $f([]) = e$ by homomorphism properties of f. Therefore, $(f \circ \eta_A)^\sharp([]) = f([])$.

Case $x :: xs$:

$$\begin{aligned}
& (f \circ \eta_A)^\sharp(x :: xs) & \\
&= (f \circ \eta_A)(x) \bullet (f \circ \eta_A)^\sharp(xs) & \text{by definition of } (-)^\sharp \\
&= (f \circ \eta_A)(x) \bullet f(xs) & \text{by induction} \\
&= f([x]) \bullet f(xs) & \text{by definition of } \eta_A \\
&= f([x] \mathbin{+\!\!+} xs) & \text{by homomorphism properties of } f \\
&= f(x :: xs) & \text{by definition of concatenation}
\end{aligned}$$

By function extensionality, $(-)^\sharp \circ (- \circ \eta_A) = id$. Therefore, $(-)^\sharp$ and $(-) \circ [\_]$ are inverse of each other.

We have now shown that $(-) \circ \eta_A$ is an equivalence from monoid homomorphisms $\mathsf{List}(A) \to \mathfrak{X}$ to set functions $A \to X$, and its inverse is given by $(-)^\sharp$, which maps set functions $A \to X$ to monoid homomorphisms $\mathsf{List}(A) \to \mathfrak{X}$. Therefore, List is indeed the free monoid. □

Lemma 1. *Zero-length arrays $(0, f)$ are contractible.*

Proof. We need to show $f : \mathsf{Fin}_0 \to A$ is equal to $\lambda\{\}$. By function extensionality this amounts to showing for all $x : \mathbf{0}$, $f(x) = (\lambda\{\})(x)$, which holds by absurdity elimination on x. Therefore, any array $(0, f)$ is equal to $(0, \lambda\{\})$. □

Proposition 5. *$(\mathsf{Array}(A), \mathbin{+\!\!+})$ is a monoid.*

Proof. To show Array satisfies left unit, we want to show $(0, \lambda\{\}) +\!\!+ (n, f) = (n, f)$.

$$(0, \lambda\{\}) +\!\!+ (n, f) = (0 + n, \lambda\{\} \oplus f)$$

$$(\lambda\{\} \oplus f)(k) = \begin{cases} (\lambda\{\})(k) & \text{if } k < 0 \\ f(k-0) & \text{otherwise} \end{cases}$$

It is trivial to see the length matches: $0 + n = n$. We also need to show $\lambda\{\} \oplus f = f$. Since $n < 0$ for any $n : \mathbb{N}$ is impossible, $(\lambda\{\} \oplus f)(k)$ would always reduce to $f(k-0) = f(k)$, therefore $(0, \lambda\{\}) +\!\!+ (n, f) = (n, f)$.

To show Array satisfies right unit, we want to show $(n, f) +\!\!+ (0, \lambda\{\}) = (n, f)$.

$$(n, f) +\!\!+ (0, \lambda\{\}) = (n + 0, f \oplus \lambda\{\})$$

$$(f \oplus \lambda\{\})(k) = \begin{cases} f(k) & \text{if } k < n \\ (\lambda\{\})(k-0) & \text{otherwise} \end{cases}$$

It is trivial to see the length matches: $n + 0 = n$. We also need to show $f \oplus \lambda\{\} = f$. We note that the type of $f \oplus \lambda\{\}$ is $\mathsf{Fin}_{\mathsf{n+0}} \longrightarrow A$, therefore k is of the type $\mathsf{Fin}_{\mathsf{n+0}}$. Since $\mathsf{Fin}_{\mathsf{n+0}} \cong \mathsf{Fin}_{\mathsf{n}}$, it must always hold that $k < n$, therefore $(f \oplus \lambda\{\})(k)$ must always reduce to $f(k)$. Thus, $(n, f) +\!\!+ (0, \lambda\{\}) = (n, f)$.

For associativity, we want to show for any array (n, f), (m, g), (o, h), $((n, f) +\!\!+ (m, g)) +\!\!+ (o, h) = (n, f) +\!\!+ ((m, g) +\!\!+ (o, h))$.

$$((n, f) +\!\!+ (m, g)) +\!\!+ (o, h) = ((n + m) + o, (f \oplus g) \oplus h)$$

$$((f \oplus g) \oplus h)(k) = \begin{cases} \begin{cases} f(k) & \text{if } k < n \\ g(k-n) & \text{otherwise} \end{cases} & \text{if } k < n + m \\ h(k - (n+m)) & \text{otherwise} \end{cases}$$

$$(n, f) +\!\!+ ((m, g) +\!\!+ (o, h)) = (n + (m + o), f \oplus (g \oplus h))$$

$$(f \oplus (g \oplus h))(k) = \begin{cases} f(k) & \text{k < n} \\ \begin{cases} g(k-n) & \text{k - n < m} \\ h(k-n-m) & \text{otherwise} \end{cases} & \text{otherwise} \end{cases}$$

We first case split on $k < n + m$ then $k < n$.

Case $k < n + m$, $k < n$: Both $(f \oplus (g \oplus h))(k)$ and $((f \oplus g) \oplus h)(k)$ reduce to $f(k)$.

Case $k < n + m$, $k \geq n$: $((f \oplus g) \oplus h)(k)$ reduce to $g(k-n)$ by definition. To show $(f \oplus (g \oplus h))(k)$ would also reduce to $g(k-n)$, we first need to show $\neg(k < n)$, which follows from $k \geq n$. We then need to show $k - n < m$. This can be done by simply subtracting n from both side on $k < n + m$, which is well defined since $k \geq n$.

Case $k \geq n + m$: $((f \oplus g) \oplus h)(k)$ reduce to $h(k - (n+m))$ by definition. To show $(f \oplus (g \oplus h))(k)$ would also reduce to $h(k - (n+m))$, we first need to show $\neg(k < n)$, which follows from $k \geq n + m$. We then need to show $\neg(k - n < m)$, which also follows from $k \geq n + m$. We now have $(f \oplus (g \oplus h))(k) = h(k - n - m)$. Since $k \geq n + m$, $h(k-n-m)$ is well defined and is equal to $h(k - (n+m))$, therefore $(f \oplus (g \oplus h))(k) = (f \oplus g) \oplus h)(k) = h(k - (n+m))$.

In all cases $(f \oplus (g \oplus h))(k) = ((f \oplus g) \oplus h)(k)$, therefore associativity holds. □

Lemma 2 (Array cons). *Any array $(S(n), f)$ is equal to $\eta_A(f(0)) +\!\!+ (n, f \circ S)$.*

Proof. We want to show $\eta_A(f(0)) +\!\!+ (n, f \circ S) = (S(n), f)$.

$$(1, \lambda\{0 \mapsto f(0)\}) +\!\!+ (n, f \circ S) = (1 + n, \lambda\{0 \mapsto f(0)\} \oplus (f \circ S))$$

$$(\lambda\{0 \mapsto f(0)\} \oplus (f \circ S))(k) = \begin{cases} f(0) & \text{if } k < 1 \\ (f \circ S)(k-1) & \text{otherwise} \end{cases}$$

It is trivial to see the length matches: $1 + n = S(n)$. We need to show $(\lambda\{0 \mapsto f(0)\} \oplus (f \circ S)) = f$. We prove by case splitting on $k < 1$. On $k < 1$, $(\lambda\{0 \mapsto f(0)\} \oplus (f \circ S))(k)$ reduces to $f(0)$. Since, the only possible for k when $k < 1$ is 0, $(\lambda\{0 \mapsto f(0)\} \oplus (f \circ S))(k) = f(k)$ when $k < 1$. On $k \geq 1$, $(\lambda\{0 \mapsto f(0)\} \oplus (f \circ S))(k)$ reduces to $(f \circ S)(k-1) = f(S(k-1))$. Since $k \geq 1$, $S(k-1) = k$, therefore $(\lambda\{0 \mapsto f(0)\} \oplus (f \circ S))(k) = f(k)$ when $k \geq 1$. Thus, in both cases, $(\lambda\{0 \mapsto f(0)\} \oplus (f \circ S)) = f$. □

Lemma 3 (Array split)**.** *For any array* $(S(n), f)$ *and* (m, g),

$$(n + m, (f \oplus g) \circ S) = (n, f \circ S) +\!\!+ (m, g) \quad .$$

Proof. It is trivial to see both array have length $n + m$. We want to show $(f \oplus g) \circ S = (f \circ S) \oplus g$.

$$((f \oplus g) \circ S)(k) = \begin{cases} f(S(k)) & \text{if } S(k) < S(n) \\ g(S(k) - S(n)) & \text{otherwise} \end{cases}$$

$$((f \circ S) \oplus g)(k) = \begin{cases} (f \circ S)(k) & \text{if } k < n \\ g(k - n) & \text{otherwise} \end{cases}$$

We prove by case splitting on $k < n$. On $k < n$, $((f \oplus g) \circ S)(k)$ reduces to $f(S(k))$ since $k < n$ implies $S(k) < S(n)$, and $((f \circ S) \oplus g)(k)$ reduces to $(f \circ S)(k)$ by definition, therefore they are equal. On $k \geq n$, $((f \oplus g) \circ S)(k)$ reduces to $g(S(k) - S(n)) = g(k - n)$, and $((f \circ S) \oplus g)(k)$ reduces to $g(k - n)$ by definition, therefore they are equal. □

Proposition 6. $(-)^\sharp$ *lifts a function* $f : A \to X$ *to a monoid homomorphism* $f^\sharp : \mathsf{Array}(A) \to \mathfrak{X}$.

Proof. To show that $f^\sharp$ is a monoid homomorphism, we need to show $f^\sharp(xs +\!\!+ ys) = f^\sharp(xs) \bullet f^\sharp(ys)$. We can do so by induction on xs.

Case $(0, g)$: We have $g = \lambda\{\}$ by Lemma 1. $f^\sharp((0, \lambda\{\}) +\!\!+ ys) = f^\sharp(ys)$ by left unit, and $f^\sharp(0, \lambda\{\}) \bullet f^\sharp(ys) = e \bullet f^\sharp(ys) = f^\sharp(ys)$ by definition of $(-)^\sharp$. Therefore, $f^\sharp((0, \lambda\{\}) +\!\!+ ys) = f^\sharp(0, \lambda\{\}) \bullet f^\sharp(ys)$.

Case $(S(n), g)$: Let ys be (m, h).

$$\begin{aligned}
& f^\sharp((S(n), g) +\!\!+ (m, h)) \\
&= f^\sharp(S(n + m), g \oplus h) && \text{by definition of concatenation} \\
&= f((g \oplus h)(0)) \bullet f^\sharp(n + m, (g \oplus h) \circ S) && \text{by definition of } (-)^\sharp \\
&= f(g(0)) \bullet f^\sharp(n + m, (g \oplus h) \circ S) && \text{by definition of } \oplus \text{, and } 0 < S(n) \\
&= f(g(0)) \bullet f^\sharp((n, g \circ S) +\!\!+ (m, h)) && \text{by Lemma 3} \\
&= f(g(0)) \bullet (f^\sharp(n, g \circ S) \bullet f^\sharp(m, h)) && \text{by induction} \\
&= (f(g(0)) \bullet f^\sharp(n, g \circ S)) \bullet f^\sharp(m, h)) && \text{by associativity} \\
&= f^\sharp(S(n), g) \bullet f^\sharp(m, h)) && \text{by definition of } (-)^\sharp
\end{aligned}$$

Therefore, $(-)^\sharp$ does correctly lift a function to a monoid homomorphism. □

Proposition 7 (Universal property for Array)**.** $(\mathsf{Array}(A), \eta_A)$ *is the free monoid on* A.

Proof. To show that $(-)^\sharp$ is an inverse to $- \circ \eta_A$, we first show $(-)^\sharp$ is the right inverse to $- \circ \eta_A$. For all f and x, $(f^\sharp \circ \eta_A)(x) = f^\sharp(1, \lambda\{0 \mapsto x\}) = f(x) \bullet e = f(x)$, therefore by function extensionality, for any f, $f^\sharp \circ \eta_A = f$, and $(- \circ \eta_A) \circ (-)^\sharp = id$.

To show $(-)^\sharp$ is the left inverse to $- \circ \eta_A$, we need to prove for any monoid homomorphism $f : \mathsf{Array}(A) \to \mathfrak{X}$, $(f \circ \eta_A)^\sharp(xs) = f(xs)$. We can do so by induction on xs.

Case $(0, g)$: By Lemma 1 we have $g = \lambda\{\}$. $(f \circ \eta_A)^\sharp(0, \lambda\{\}) = e$ by definition of the $(-)^\sharp$ operation, and $f(0, \lambda\{\}) = e$ by homomorphism properties of f. Therefore, $(f \circ \eta_A)^\sharp(0, g) = f(0, g)$.

Case $(S(n), g)$, we prove it in reverse:

$$\begin{aligned}
& f(S(n), g) \\
&= f(\eta_A(g(0)) +\!\!+ (n, g \circ S)) && \text{by Lemma 2} \\
&= f(\eta_A(g(0))) \bullet f(n, g \circ S) && \text{by homomorphism properties of } f \\
&= (f \circ \eta_A)(g(0)) \bullet (f \circ \eta_A)^\sharp(n, g \circ S) && \text{by induction} \\
&= (f \circ \eta_A)^\sharp(S(n), g) && \text{by definition of } (-)^\sharp
\end{aligned}$$

By function extensionality, $(-)^\sharp \circ (- \circ \eta_A) = id$. Therefore, $(-)^\sharp$ and $(-) \circ [\_]$ are inverse of each other.

We have now shown that $(-) \circ \eta_A$ is an equivalence from monoid homomorphisms $\mathsf{Array}(A) \to \mathfrak{X}$ to set functions $A \to X$, and its inverse is given by $(-)^\sharp$, which maps set functions $A \to X$ to monoid homomorphisms $\mathsf{Array}(A) \to \mathfrak{X}$. Therefore, Array is indeed the free monoid. ◻

D Supplementary material for Section 5 (Constructions of Free Comm. Monoids)

Proposition 8. *$(\mathfrak{F}(A) /\!\!/ \approx, \bullet, q(e))$ is a commutative monoid.*

Proof. Since $\approx$ is a congruence wrt $\bullet$, we can lift $\bullet : F(A) \to F(A) \to F(A)$ to the quotient to obtain $+\!\!+ : F(A) /\!\!/ \approx \to F(A) /\!\!/ \approx \to F(A) /\!\!/ \approx$. $+\!\!+$ also satisfies the unit and associativity laws that $\bullet$ satisfy. Commutativity of $+\!\!+$ follows from the commutativity requirement of $\approx$, therefore $(F(A) /\!\!/ \approx, +\!\!+, q(i))$ forms a commutative monoid. ◻

Proposition 9 (Universal property for $\mathfrak{F}(A) /\!\!/ \approx$)**.** *$(\mathfrak{F}(A) /\!\!/ \approx, \eta_A : A \xrightarrow{\eta_A} \mathfrak{F}(A) \xrightarrow{q} \mathfrak{F}(A) /\!\!/ \approx)$ is the free comm. monoid on A.*

Proof. To show that $(-)^\sharp$ is an inverse to $(-) \circ \eta_A$, we first show $(-)^\sharp$ is the right inverse to $(-) \circ \eta_A$. For all f and x, $(f^\sharp \circ \eta_A)(x) = (f^\sharp \circ q)(\mu_A(x)) = \widehat{f}(\mu_A(x))$. By universal property of F, $\widehat{f}(\mu_A(x)) = f(x)$, therefore $(f^\sharp \circ \eta_A)(x) = f(x)$. By function extensionality, for any f, $f^\sharp \circ \eta_A = f$, and $(- \circ \eta_A) \circ (-)^\sharp = id$.

To show $(-)^\sharp$ is the left inverse to $(-) \circ \eta_A$, we need to prove for any commutative monoid homomorphism $f : \mathfrak{F}(A) \approx \to \mathfrak{X}$ and $x : \mathfrak{F}(A) \approx$, $(f \circ \eta_A)^\sharp(x) = f(x)$. To prove this it is suffice to show for all $x : \mathfrak{F}(A)$, $(f \circ \eta_A)^\sharp(q(x)) = f(q(x))$. $(f \circ \eta_A)^\sharp(q(x))$ reduces to $(\widehat{f \circ q \circ \mu_A})(x)$. Note that both f and q are homomorphism, therefore $f \circ q$ is a homomorphism. By universal property of F we get $(\widehat{f \circ q \circ \mu_A})(x) = (f \circ q)(x)$, therefore $(f \circ \eta_A)^\sharp(q(x)) = f(q(x))$.

We have now shown that $(-) \circ \eta_A$ is an equivalence from commutative monoid homomorphisms $\mathfrak{F}(A) /\!\!/ \approx \to \mathfrak{X}$ to set functions $A \to X$, and its inverse is given by $(-)^\sharp$, which maps set functions $A \to X$ to commutative monoid homomorphisms $\mathfrak{F}(A) /\!\!/ \approx \to \mathfrak{X}$. Therefore, $\mathfrak{F}(A) /\!\!/ \approx$ is indeed the free commutative monoid on A. ◻

Proposition 10. *Let $\mathfrak{X}$ be a commutative monoid, and $f : A \to X$. For $x, y : A$ and $xs, ys : \mathsf{PList}(A)$, $f^\sharp(xs +\!\!+ x :: y :: ys) = f^\sharp(xs +\!\!+ y :: x :: ys)$. Hence,* Perm *respects $(-)^\sharp$.*

Proof. We can prove this by induction on xs. For $xs = []$, by homomorphism properties of $f^\sharp$, we can prove $f^\sharp(x :: y :: ys) = f^\sharp([x]) \bullet f^\sharp([y]) \bullet f^\sharp(ys)$. Since the image of $f^\sharp$ is a commutative monoid, we have $f^\sharp([x]) \bullet f^\sharp([y]) = f^\sharp([y]) \bullet f^\sharp([x])$, therefore proving $f^\sharp(x :: y :: ys) = f^\sharp(y :: x :: ys)$. For $xs = z :: zs$, we can prove $f^\sharp((z :: zs) +\!\!+ x :: y :: ys) = f^\sharp([z]) \bullet f^\sharp(zs +\!\!+ x :: y :: ys)$. We can then complete the proof by induction to obtain $f^\sharp(zs +\!\!+ x :: y :: ys) = f^\sharp(zs +\!\!+ y :: x :: ys)$ and reassembling back to $f^\sharp((z :: zs) +\!\!+ y :: x :: ys)$ by homomorphism properties of $f^\sharp$. ◻

Also, whenever we define a function on PList by pattern matching we would also need to show the function respects Perm, i.e. $\mathsf{Perm}\, as\, bs \to f(as) = f(bs)$. This can be annoying because of the many auxiliary variables in the constructor perm-swap, namely xs, ys, zs. We need to show f would respect a swap in the list anywhere between xs and ys, which can unnecessarily complicate the proof. For example in the inductive step of Proposition 10, $f^\sharp((z :: zs) +\!\!+ x :: y :: ys) = f^\sharp([z]) \bullet f^\sharp(zs +\!\!+ x :: y :: ys)$, to actually prove this in Cubical Agda would involve first applying associativity to prove $(z :: zs) +\!\!+ x :: y :: ys = z :: (zs +\!\!+ x :: y :: ys)$, before we can actually apply homomorphism properties of f. In the final reassembling step, similarly, we also need to re-apply associativity to go from $z :: (zs +\!\!+ y :: x :: ys)$ to $(z :: zs) +\!\!+ y :: x :: ys$. Also since we are working with an equivalence relation we defined (Perm) and not the equality type directly, we cannot exploit the many combinators defined in the standard library for the equality type and often needing to re-define combinators ourselves. The trunc constructor is necessary to truncate SList down to a set, thereby ignoring any higher paths introduced by the swap constructor. This is opposed to List, which does not need a trunc constructor because it does not have any path constructors, therefore it can be proven that $\mathsf{List}(A)$ is a set assuming A is a set (see formalization).

Definition D.1 (Concatenation)**.** We define the concatenation operation $+\!\!+ : \mathsf{SList}(A) \to \mathsf{SList}(A) \to \mathsf{SList}(A)$ recursively, where we also have to consider the (functorial) action on the swap path using ap.

$$[] +\!\!+ ys = ys$$
$$(x :: xs) +\!\!+ ys = x :: (xs +\!\!+ ys)$$
$$\mathsf{ap}_{+\!\!+ ys}(\mathsf{swap}(x, y, xs)) = \mathsf{swap}(x, y, ys +\!\!+ xs)$$

[Choudhury and Fiore 2023] have already given a proof of $(\mathsf{SList}(A), +\!\!+, [])$ satisfying commutative monoid laws. We explain the proof of $+\!\!+$ satisfying commutativity here.

Lemma 8 (Head rearrange)**.** *For all $x : A$, $xs : \mathsf{SList}(A)$, $x :: xs = xs +\!\!+ [x]$.*

Proof. We can prove this by induction on xs. For $xs \equiv []$ this is trivial. For $xs \equiv y :: ys$, we have the induction hypothesis $x :: ys = ys ++ [x]$. By applying $y :: (-)$ on both side and then apply swap, we can complete the proof. □

Theorem D.2 (Commutativity)**.** *For all $xs, ys\colon$ SList(A), $xs ++ ys = ys ++ xs$.*

Proof. By induction on xs we can iteratively apply Lemma 8 to move all elements of xs to after ys. This would move ys to the head and xs to the end, thereby proving $xs ++ ys = ys ++ xs$. □

Unlike PList which is defined as a set quotient, this is defined as a HIT, therefore handling equalities between SList is much simpler than PList. We would still need to prove a function f respects the path constructor of SList when pattern matching, i.e. $f(x :: y :: xs) = f(y :: x :: xs)$. Unlike PList we do not need to worry about as many auxiliary variables since swap only happens at the head of the list, whereas with PList we would need to prove $f(xs ++ x :: y :: ys) = f(xs ++ y :: x :: ys)$. One may be tempted to just remove xs from the perm-swap constructor such that it becomes perm-swap: $\forall x\, y\, ys\, zs \to \mathsf{Perm}\,(x :: y :: ys)\, zs \to \mathsf{Perm}\,(y :: x :: ys)\, zs$. However this would break Perm's congruence wrt to $++$, therefore violating the axioms of permutation relations. Also, since we are working with the identity type directly, properties we would expect from swap, such as reflexivity, transitivity, symmetry, congruence and such all comes directly by construction, whereas with Perm we would have to prove these properties manually. We can also use the many combinatorics defined in the standard library for equational reasoning, making the handling of SList equalities a lot simpler.

Proposition 11. *$\approx$ is a equivalence relation.*

Proof. We can show any array xs is related to itself by the identity isomorphism, therefore $\approx$ is reflexive. If $xs \approx ys$ by σ, we can show $ys \approx xs$ by σ^{-1}, therefore $\approx$ is symmetric. If $xs \approx ys$ by σ and $ys \approx zs$ by ϕ, we can show $xs \approx zs$ by $\sigma \circ \phi$, therefore $\approx$ is transitive. □

On a more technical note, since Array and Bag are not simple data types, the definition of the monoid operation on them $++$ are necessarily more complicated, and unlike List, PList and SList, constructions of Array and Bag via $++$ often would not normalize into a very simple form, but would instead expand into giant trees of terms. This makes it such that when working with Array and Bag we need to be very careful or otherwise Agda would be stuck trying to display the normalized form of Array and Bag in the goal and context menu. Type-checking also becomes a lengthy process that tests if the user possesses the virtue of patience.

However, performing arbitrary partitioning with Array and Bag is much easier than List, SList, PList. For example, one can simply use the combinator $\mathsf{Fin}_{\mathsf{n+m}} \xrightarrow{\sim} \mathsf{Fin}_{\mathsf{n}} + \mathsf{Fin}_{\mathsf{m}}$ to partition the array, then perform operations on the partitions such as swapping in Proposition 13, or perform operations on the partitions individually such as two individual permutation in Proposition 12. This makes it such that when defining divide-and-conquer algorithms such as merge sort, Bag and Array are more natural to work with than List, SList, and PList.

We use $\wr x, y, \ldots \int$ to denote $\eta_A(x) \bullet \eta_A(y) \bullet \cdots : \mathcal{M}(A)$, and $[x, y, \ldots]$ to denote $\eta_A(x) \bullet \eta_A(y) \bullet \cdots : \mathcal{L}(A)$, or $x :: xs$ to denote $\eta_A(x) \bullet xs\colon \mathcal{L}(A)$.

E Supplementary material for Section 6 (Application: Sorting Functions)

Proposition 28. *$\preccurlyeq_s$ is not necessarily transitive.*

Proof. We give a counter-example of s that would violate transitivity. Consider this section $s\colon$ SList$(\mathbb{N}) \to$ List$(\mathbb{N})$, we can define a sort function sort: SList$(\mathbb{N}) \to$ List$(\mathbb{N})$ which sorts SList$(\mathbb{N})$ ascendingly. We can use sort to construct s.

$$s(xs) = \begin{cases} \mathsf{sort}(xs) & \text{if length}(xs) \text{ is odd} \\ \mathsf{reverse}(\mathsf{sort}(xs)) & \text{otherwise} \end{cases}$$

$$s([2,3,1,4]) = [4,3,2,1]$$

$$s([2,3,1]) = [1,2,3]$$

□

Proposition 29. *Assume A is a set with different elements, i.e. $\exists x, y\colon A.\, x \neq y$, we cannot construct a full equivalence between sections that satisfy* im-cut *and total orders on A.*

Proof. We give a counter-example of s that satisfy im-cut but is not a sort function. Consider the insertion sort function sort: $\mathcal{M}(\mathbb{N}) \to \mathcal{L}(\mathbb{N})$ parameterized by $\leq$:

$$\begin{aligned} \mathsf{reverseTail}([\,]) &= [\,] \\ \mathsf{reverseTail}(x :: xs) &= x :: \mathsf{reverse}(xs) \\ s(xs) &= \mathsf{reverseTail}(\mathsf{sort}(xs)) \\ s(\wr 2,3,1,4 \int) &= [1,4,3,2] \\ s(\wr 2,3,1 \int) &= [1,3,2] \\ s(\wr 2,3 \int) &= [2,3] \end{aligned}$$

By Proposition 22 we can use sort to construct $\preccurlyeq_s$ which would be equivalent to $\leq$. However, the $\preccurlyeq_s$ constructed by s would also be equivalent to $\leq$. This is because s sorts 2-element list correctly, despite $s \neq$ sort. Since two different sections satisfying im-cut maps to the same total order, there cannot be a full equivalence. □