University of Glasgow | School of Computing Science

Honours Individual Project Dissertation

Sorting correctly with symmetries

Wind Wong

March 22, 2024

Abstract

Sorting algorithms are among the most fundamental in computing science, and are frequently studied in relation to functional programming. There have been many investigations of the correctness of sorting algorithms in terms of total orders and category theory. We provide a new perspective on this topic by approaching it via universal algebra, viewing a sorting algorithm as a section (right inverse) to a surjective function from a free monoid to a free commutative monoid. This emphasizes the fact that introducing symmetry (the passage from free monoids to free commutative monoids) eliminates ordering, while sorting (the right inverse) recovers ordering.

Our first main contribution is a new axiomatization of correct sorting algorithms, using two axioms that are not expressed in terms of a pre-existing total order. Instead, the total order can be recovered from the definition of an algorithm that satisfies the axioms. Our second main contribution is a formlization of the informal intuition that commutative monoids are unordered lists. Additionally, we give new proofs of some standard results about constructions of free monoids and free commutative monoids. We formalize all of the theory in Cubical Agda.

Education Use Consent

Acknowledgement

I would like to express my deepest gratitude to my supervisor, Dr. Vikraman Choudhury, for their invaluable guidance and support throughout the project. Their expertise and patience in teaching me the intricacies of Cubical Type Theory have been crucial to the successful completion of this project.

I would also like to extend my heartfelt thanks to my co-supervisor Dr. Simon Gay for his guidance in refining my writing. His feedbacks have significantly improved the quality of this dissertation and his assistance in helping me articulate my thoughts more clearly and succinctly has been invaluable.

Disclaimer My supervisors and I wrote both an abstract for the HoTT/UF 2024 workshop and a paper submission for ICFP 2024. The abstract has been accepted by the workshop and the result of the ICFP submission is currently pending. The content of this dissertation is primarily copied from the content of the ICFP submission which I have authored. Content which my supervisors co-authored are rewritten by me for this submission, for the purpose of avoiding plagiarism and simplifying the technical details.

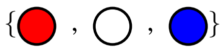
The "we" used throughout the dissertation is used in the sense of authorial "we".

Contents

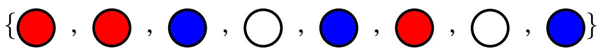
1 Introduction

1.1 Motivation

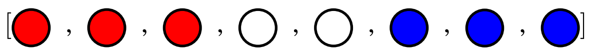
Consider a puzzle about sorting, inspired by Dijkstra's Dutch National Flag problem [Dijkstra 1997, Ch.14]. Suppose there are balls of three colors, corresponding to the colors of the Dutch flag: red, white, and blue.



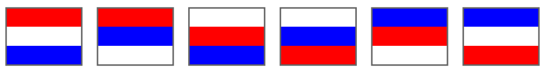
Given an unordered list of such balls, how many ways can you sort them into the Dutch flag?



Obviously there is only one way, which is given by the order $\mathsf{red} < \mathsf{white} < \mathsf{blue}$.

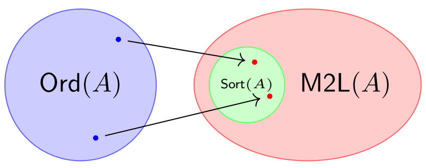


What if we are avid enjoyers of vexillology who also want to consider other flags? We might ask: how many ways can we sort our unordered list of balls? We know that there are only $3! = 6$ permutations of $\{\mathsf{red}, \mathsf{white}, \mathsf{blue}\}$, so there are only 6 possible orderings we can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



We posit that because there are exactly 6 orderings, we can only define 6 *correct* sorting functions.

In more formal terms: let $A = \{\mathsf{red}, \mathsf{white}, \mathsf{blue}\}$ let $\mathsf{Ord}(A)$ be the set of orderings on A and let $\mathsf{M2L}(A)$ be the set of functions $\mathsf{UnorderedList}(A) \to \mathsf{List}(A)$. There is a function $\mathsf{o2f} : \mathsf{Ord}(A) \to \mathsf{Sort}(A)$ that maps orderings to some subset $\mathsf{Sort}(A) \subset \mathsf{M2L}(A)$ which we call "sort functions":



This *seems* trivial to prove: we can construct $\mathsf{o2f}(r)$ by parameterizing any sorting algorithm, e.g. insertion sort, by the order relation r and we are done! This however is not enough: we want to show that there are exactly 6 sorting functions because there are exactly 6 orderings. In other words, $\mathsf{Ord}(A)$ and $\mathsf{Sort}(A)$ should be isomorphic. There should exist an inverse mapping to $\mathsf{o2f}$ that allows us to construct an ordering $\preccurlyeq$ given a function $s \in \mathsf{Sort}(A)$, such that $\forall s \in \mathsf{Sort}(A).\ \mathsf{o2f}(\mathsf{o2f}^{-1}(s)) = s \wedge \forall r \in \mathsf{Ord}(A).\ \mathsf{o2f}^{-1}(\mathsf{o2f}(r)) = r$.

To complete this proof formally we need to execute the following plan.

1. We need to formalize exactly what $\mathsf{UnorderedList}(A)$ and $\mathsf{List}(A)$ are. We formalize them in terms of universal algebra: $\mathsf{UnorderedList}(A)$ is the free commutative monoid on A and $\mathsf{List}(A)$ is the free monoid on A. We give different constructions of them in Chapter 4 and Chapter 5, proving these constructions correct by verifying their universal properties. We also explore the relationship between commutativity and ordering in Chapter 5 and Section 6.5, showing how imposing commutativity on a free monoid is same as "forgetting" the order, giving a formal notion to the idea of unordered list.
2. We need to nail down exactly what the subset $\mathsf{Sort}(A)$ is. In other words, we need to identify what properties functions $f \in \mathsf{Sort}(A) \subset \mathsf{M2L}(A)$ satisfy which separate them from other functions $g \in \mathsf{M2L}(A)$. We identify two properties in Section 7.1, giving us two axioms for sorting functions which do not assume a pre-existing total order. This distinguishes our approach from other formalizations of the correctness of sorting such as [Appel 2023].
3. Using the properties we have identified, we need to construct a full equivalence $\mathsf{Sort}(A) \simeq \mathsf{Ord}(A)$. Mapping orders to sorting functions is trivial by parameterizing insertion sort on an order relation. More interestingly we need to show how sorting functions can be mapped to orders, showing that the properties we have identified for $\mathsf{Sort}(A)$ indeed correctly identify the usual notion of sorting functions.

The term "formalize" in the above plan means exactly that, and we therefore need to choose a proof system in which to work. Martin-Löf Type Theory (MLTT) would not be convenient because of the large amount of work with set quotients and commutativity in our theory. These concepts do not behave well in MLTT and often can only be modelled via setoids. While setoids would suffice for our purposes, they carry a high proof burden which leads to a phenomenon (un)affectionately named "setoid hell". We have therefore decided to formalize our proofs for this paper in Cubical Agda [Vezzosi et al. 2019]. This allows us to work naturally with higher inductive types, and allows for a very simple method of transporting proofs between different constructions of free algebras by univalence. What also makes Cubical Agda attractive is that Cubical Type Theory [Cohen et al. 2018] allows us to enjoy the full power of univalence given by Homotopy Type Theory [Univalent Foundations Program 2013], without losing canonicity and computational content of the proofs. This means that proofs and functions transported by univalence actually compute! We give a table of our formalization in Chapter 8.

1.2 Outline and Contributions

- In Section 2.2, we describe the notation we use in the paper.
- In Chapter 3, we give some background on a categorical framework for universal algebra including equations, and its formalization in HoTT. We give the definition of free algebras and their universal property.
- In Chapter 4, we give various constructions of free monoids, and their proofs of universal property.
- In Chapter 5, we add commutativity to free monoids, and show how to extend the proofs of universal property appropriately. Using our results we prove existing constructions of free commutative monoid are correct directly by their universal property, as opposed to the more common method of proving their correctness by isomorphism to known constructions of free commutative monoid.
- In Chapter 6, we show how free monoids and free commutative monoids are different, by discussing various combinatorial properties and operations, which can be defined for both, and ones which cannot be defined for both.
- In Section 7.1, we build on the constructions of the previous sections and study intrinsically verified sorting function. The main result in this section is to connect total orders and sorting and commutativity, by proving an equivalence between decidable total orders on a carrier set A, and correct sorting algorithms on lists of elements of A.

- All the work in this paper is formalized in Cubical Agda, which is discussed in Chapter 8, and the accompanying code is available as supplementary material. Although the formalization is a contribution in itself, the purpose of the paper is not to discuss the formalization, but to present the results in un-formalized form, using HoTT/UF as a foundation, so the ideas are accessible to a wider audience, even without a background in type theory or formalization.
- In Chapter 9, we discuss connections to related work, and future work.

1.3 Navigation

For ease of navigating the Agda formalization code, definitions and theorems stated in this dissertation might have (link) next to it, which is a hyperlink pointing to a HTML rendering of the relevant Agda code. The Agda code is also submitted along the dissertation, but it may be difficult to navigate since as plain source code since without the HTML clickable elements one cannot easily go to functions and definitions defined in another file within the source code.

2 Type Theory

2.1 Type Theory

As we have explained, our work is formalized in Cubical Agda and Cubical Type Theory, which is a variant of Homotopy Type Theory that is designed to preserve computational properties of type theory. We refer the readers to other works such as [Vezzosi et al. 2019] and [Cohen et al. 2018] for a more in-depth explanation on Cubical Type Theory and how we can program in Cubical Agda. We also give an overview of relevant features in Cubical Type Theory which normal type theory (namely MLTT) lacks, and how these features are used within the scope of our work.

2.1.1 Dependent Types

We first review some basic ideas from type theory. We assume the readers have experience with dependent typed programming, so this would only serves as a refresher.

Π-types Π-types generalizes function types. These types capture the idea of functions that can take different types as input or return different types as output depending on the input. The type of a function that accepts an argument $x : A$ and returns an element of type $B(x)$, where B might depend on x, can be expressed as $(x : A) \to B(x)$ or $\Pi_{(x:A)}B(x)$.

Σ-types Σ-types generalizes product types. These types represent tuples where the type of the second component depends on the first. An element of the type $\Sigma_{(x:A)}B(x)$ is a pair (a, b) where a is of type A and b is of type $B(a)$.

Identity types The identity type of a type A expresses equality in type theory. If a and b are elements of type A, then the type expressing that a is equal to b is often written $a =_A b$. Optionally, the type can be omitted for brevity, therefore written just as $a = b$. This is often useful to state when two elements should be equal but they are not definitionally equal. For example, given $n : \mathbb{N}$, the elements $n+0 : \mathbb{N}$ and $0+n : \mathbb{N}$ should be the same, however depending on the implementation of $+$ they may not be definitionally equal, therefore $B(n+0)$ and $B(0+n)$ would give us different types definitionally. Having a type $n + 0 =_{\mathbb{N}} 0 + n$ would allow us to convert $x : B(n + 0)$ to $x : B(0 + n)$ and vice versa, therefore solving the over-restriction of definitional equality.

Propositions as Types The Curry-Howard correspondence gives us the propositions-as-types principle, which states that propositional logic can be interpreted as types:

- A type P can be viewed as a proposition, and a term of that type is a proof of the proposition.
- Π-types correspond to universal quantification or implications. A function $f : P \to Q$ can be thought of as "P implies Q", and a function $f : (a : A) \to P(a)$ can be thought of as "for all a of type A, $P(a)$ holds".
- Σ-types correspond to existential quantification. An element of $\Sigma_(a : A)\, P(a)$ can be thought of as a proof that states "there exists an a of type A such that $P(a)$ holds".

2.1.2 Homotopy Types

In Homotopy Type Theory, types are assigned a homotopy level (often abbreviated h-level). We explain the concept of h-level since it is important to understand the terminlogy used in the rest of the dissertation.

Contractible Type A type A is said to be contractible iff there is an element $a : A$, refered to as the *center of contraction*, such that for all $x : A$, we have $a = x$. More formally:

$$\mathsf{isContr}(A) :\equiv \Sigma_{(a:A)}\Pi_{(x:A)}(a =_A x).$$

Contractible types are assigned the h-level 0. One example of such a type would be the unit type $\mathbf{1}$. In fact, all contractible types are equivalent to $\mathbf{1}$.

Mere proposition Going one dimension higher, we have mere propositions. A type A is said to be a mere proposition iff all elements of A are equal. We give two formal and equivalent definitions:

$$\mathsf{isProp}(A) :\equiv \Pi_{(x,y:A)}(x =_A y).$$
$$\mathsf{isProp}(A) :\equiv \Pi_{(x,y:A)}\ \mathsf{isContr}(x =_A y).$$

Mere propositions are assigned the h-level 1. We use $\mathsf{hProp} :\equiv \Sigma_{\mathcal{U}}\ \mathsf{isProp}$. to denote the type for all types that are propositions. Examples of such types are the unit type $\mathbf{1}$ and the empty type $\mathbf{0}$. In HoTT, we use the adverb *merely* to describe propositions represented as a mere propositional type. We also introduce the idea of propositional truncation, where a type A is truncated to a mere propositional type $\|A\|$. This can be done with a higher inductive type (elaborated more in Section 2.1.4):

```
data ‖_‖ (A : 𝒰) : 𝒰 where
    [_]  : A → ‖A‖
    trunc : (x, y : ‖A‖) → x = y
```

One example usage of propositional truncation is the definition of *mere existence*, where $\exists (x : A).\, P(x)$ is represented as $\left\|\Sigma_{(x:A)}P(x)\right\|$. This differs from $\Sigma_{(x:A)}P(x)$ in that mere existence only contains information whether such an x that satifies P exists, whereas the Σ type contains more information, namely the value of the actual element x. This makes it such that with a mere existential proof of x we would only be able to use x to prove other mere propositions, but we would not be able to define any functions that actually depend on the value of x. For example, consider a function $f : (x : A) \to P(x) \to \mathbb{N}$ which takes an $x : A$ that satisfies P and return a $\mathbb{N}$. Given $\exists x.\, P(x)$ we would not be able to apply f on x, since $\mathbb{N}$ is not a proposition. On the other hand, assuming we have another mere proposition $Q : A \to \mathsf{hProp}$, and a proof that $\forall y.\, P(y) \to Q(y)$, we can use $\exists x.\, P(x)$ to prove $\exists x.\, Q(x)$.

Sets Going one dimension higher, we have sets. A type A is said to be a set iff the identity type of A is a mere proposition.

$$\mathsf{isSet}(A) :\equiv \Pi_{(x,y:A)}\ \mathsf{isProp}(x =_A y).$$

Sets are assigned the h-level 2. We use $\mathsf{hSet} :\equiv \Sigma_{\mathcal{U}}\ \mathsf{isSet}$. to denote the type for all types that are sets. Examples of sets would be $\mathbb{N}$, the boolean type $\mathbf{2}$, and also the unit and empty type $\mathbf{1}$ and $\mathbf{0}$. The type for all types that are mere propositions hProp is also a set.

We note that in type theories with K-axiom or uniqueness of identity proofs (UIP), all types are sets. For example, in Idris2 [Brady 2021], we can prove this by pattern matching on proofs:

```
uip : (A : Type) → (x, y : A) → (p, q : x = y) → p = q
uip A x x Refl Refl = Refl
```

The K elimination rule allows us to pattern match on identity types and force them to be Refl, thus allowing us to prove $p = q$ since both p and q are Refl. HoTT, however, does not assume the K-axiom, therefore this would be invalid in HoTT.

In the dissertation we would be primarily working with sets (and mere propositions since all mere propositions are sets). There are higher dimensional types, but generalizing the work to higher dimensions is currently left for future works.

Groupoid Going one dimension higher, we have groupoids. A type A is said to be a groupoid iff the identity type of A is a set.

$$\mathsf{isGroupoid}(A) :\equiv \Pi_{(x,y:A)}\, \mathsf{isSet}(x =_A y).$$

Groupoids are assigned the h-level 3. All sets are groupoids, and the type for all types that are sets hSet is also a groupoid. For example, consider the boolean type $\mathbf{2} : \mathsf{hSet}$. There are two different elements of $\mathbf{2} =_{\mathsf{hSet}} \mathbf{2}$, which generated by univalence (elaborated more in Section 2.1.5) from two different equivalence we can define on $\mathbf{2}$ to $\mathbf{2}$, the identity function id and the negate functon not. This shows that $\mathbf{2} =_{\mathsf{hSet}} \mathbf{2}$ is *not* a mere proposition, therefore hSet is *not* a set. This shows how K-axiom is incompatible with univalence and HoTT.

n-Types We can define further higher dimensional types by induction, giving us 2-groupoids, 3-groupoids, and so on. More generally, we can define n-types as below, with n starting at -2.

$$\mathsf{is\text{-}n\text{-}type}(A) :\equiv \begin{cases} \mathsf{isContr}(A) & \text{if } n = -2 \\ \prod_{x,y:A} \mathsf{is\text{-}(n\text{-}1)\text{-}type}(x =_A y) & \text{otherwise} \end{cases}$$

Somewhat confusingly, (-2)-types (contractible types) have h-level 0, and (-1)-types (mere propositions) have h-level 1, and so on. In the lingo of HoTT, when we say n-types we start counting from -2, but for h-levels we start counting from 0. This is because when the concept of h-level is invented Voevodsky chose to start counting from 0 [Voevodsky 2010], but the idea of n-trunction originates earlier from inf-category theory, where truncation level starts from -2 [Lurie 2008].

2.1.3 Function Extensionality

Function extensionality funExt states that given two functions f and g, $\mathsf{funExt} : \forall x.\, f(x) = g(x) \to f = g$. MLTT by itself does not have funExt, instead it has to be postulated as an axiom, thereby losing canonicity, in other words, we would not be able to compute any constructions of elements that involve funExt in its construction. In Cubical Type Theory, we can derive funExt as a theorem while also preserving canoncity, therefore we can compute with constructions involving funExt!

Within the scope of our work, funExt is heavily used in Section 4.2 and Section 5.4, where a n-element array A^n is defined as lookup functions $\mathsf{Fin}_n \to A$. Therefore, to prove two arrays are equal, we need to show that two functions would be equal, which is impossible to do without funExt. We also would not be able to normalize any constructions of arrays which involve funExt if it is postulated as an axiom, therefore the computational property of Cubical Type Theory is really useful for us.

2.1.4 Higher Inductive Types

Higher inductive types allow us to extend inductive types to not only allow point constructors but also path constructors, essentially equalities between elements of the HIT. One such example would be the following definition of $\mathbb{Z}$:

```
data ℤ : 𝓤 where
    pos : (n : ℕ) → ℤ
    neg: (n : ℕ) → ℤ
    posneg : pos 0 ≡ neg 0
```

The integers are often represented with a $\mathsf{pos} : \mathbb{N} \to \mathbb{Z}$ and $\mathsf{negsuc} : \mathbb{N} \to \mathbb{Z}$, where natural numbers are mapped into the integers via the pos constructor, and negative numbers are constructed by mapping $n : \mathbb{N}$ to $-(n+1)$. The shifting by one is done to avoid having duplicate elements for 0, which can easily lead to confusion. HIT allows us to define integers more naturally by saying $\mathsf{pos}(0) = \mathsf{neg}(0)$, avoiding the confusing shift-by-one hack.

With HITs we can define a general data type for set quotients:

```
data _/_ (A : 𝓤) (R : A → A → 𝓤) : 𝓤 where
    [_]  : A → A / R
    q/ : (a b : A) → (r : R a b) → [ a ] ≡ [ b ]
    trunc : (x y : A / R) → (p q : x ≡ y) → p ≡ q
```

The q/ constructor says if a and b can be identified by a relation R, then they should belong to the same quotient generated by R. One might also notice the trunc constructor, which says and proofs of $x =_{A/R} y$ are equal. This constructor basically truncates the _/_ type down to set, so that we do not have to concern ourselves with higher paths generated by the q/ constructor. We give a more precise definition of "set" in Homotopy Type Theory and examples of types that are higher dimension than sets in Section 2.1.5.

In our work, higher inductive types and set quotients are used extensively to define commutative data structures, which we would demonstrate in Chapter 5. In MLTT we can only reason with quotients and commutativity by setoids, which comes with a lot of proof burden since we cannot work with the type theory's definition of equalities and functions directly, instead we have to define our own equality relation and define our own type for setoid homomorphisms, giving rise to the infamous setoid hell.

2.1.5 Univalence

In MLTT we cannot construct equalities between types. Univalence, the core of Homotopy Type Theory, gives us equalities between types by the univalence axiom. To see how this is useful, consider $A, B : \mathcal{U}, P : \mathcal{U} \to \mathcal{U}, A = B$, we can get $P(B)$ from $P(A)$ by transport (or substitution).

To define univalence, we first need to define the notion of equivalence:

Definition 2.1.1 (Equivalence). *A function $f : A \to B$ is said to be an equivalence function iff it has a left inverse $g : B \to A$ and right inverse $h : B \to A$ such that $(\forall x.\, g(f(x)) = x) \wedge (\forall x.\, f(h(x)) = x)$. Two types A and B are said to be equivalent iff there exists an equivalence function $f : A \to B$.*

We note that the definition of equivalence is very similar to an isomorphism, only differing in that an equivalence can have different functions for left and right inverse, whereas an isomorphism would have the same function, therefore making it just an (ordinary) inverse.

Assuming A (and subsequently B by equivalence) is a hSet, f being a equivalence function would imply it is an isomorphism and vice versa, thus making both definitions the same. However, for higher dimensional types, these two definitions would not necessarily imply each other. More explicitly: let $\mathsf{qinv}(f)$ be the witness on f stating f is an isomorphism, and $\mathsf{isequiv}(f)$ be the witness on f stating f is an equivalence function.

$$\begin{aligned}\mathsf{qinv}(f) &= \Sigma_{g:B\to A}(f \circ g \sim id) \times (g \circ f \sim id)\\ \mathsf{isequiv}(f) &= (\Sigma_{g:B\to A}(f \circ g \sim id)) \times (\Sigma_{h:B\to A}(h \circ f \sim id))\end{aligned}$$

$\mathsf{isequiv}(f)$ is a mere proposition while $\mathsf{qinv}(f)$ is not necessarily a mere proposition. We refer the readers to [Univalent Foundations Program 2013, Chapter 4.1] for a proof.

Definition 2.1.2 (Univalence axiom). $(A = B) \simeq (A \simeq B)$

Essentially, if we can construct an equivalence between two types A and B, we can obtain a equality of the types A and B. While in HoTT we cannot compute with elements constructed with univalence since it is postulated as an axiom, Cubical Type Theory allows us to derive univalence as a computable theorem, therefore we can transport functions and proofs from one type to another freely without worrying about terms not being able to normalize!

Within the scope of our work, since we have multiple constructions of free monoids and free commutative monoids, given in Chapter 4 and Chapter 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 Section 6.2, where we want to show to propositions are commutative: i.e. $\forall p, q : \mathsf{hProp}, p \vee q = q \vee p$. Since p and q are types, we need univalence to show $p \vee q = q \vee p$ are in fact equal.

2.2 Notation

We give a quick overview of the notations used in this paper.

We denote the type of types with $\mathcal{U}$. In practice, $\mathcal{U}$ would be indexed by a type level à la Russell for consistency, however we opted to omit the type level for simplicity and clarity. We use $\mathsf{Fin_n}$ to denote finite sets of cardinality n in HoTT [Yorgey 2014]. This is defined as follows:

```
Fin : ℕ → 𝓤
Fin n = Σ[ m ∈ ℕ ] (m < n)
```

There are other ways to define Fin, most notably as an indexed inductive type:

```
data Fin : ℕ → 𝓤 where
    fzero : {k : ℕ} → Fin (suc k)
    fsuc : {k : ℕ} → Fin k → Fin (suc k)
```

Alternatively, it can also be defined using coproducts:

```
Fin : ℕ → 𝓤
Fin 0 = Void
Fin (suc n) = Unit + Fin n
```

In our construction we opted to use the Σ-type definition because cubical Agda does not behave well when pattern matching on indexed inductive types, and the Σ-type definition allows us to extract the underlying ℕ easily, thus allowing us to perform arithmetic which would be useful for constructions in Section 4.2 and Section 5.4.

We use Set to denote hSet while omitting the type level. We also use $\times$ to denote product types and $+$ to denote coproduct types. For mere propositions, we use $\wedge$ to denote logical and, and $\vee$ to denote logical or. In Cubical Agda these are defined as propositionally-truncated products and coproducts. We also use $\exists$ to denote mere existence, which is defined as propositionally-truncated Σ-types are shown above.

3 Universal Algebra

As explained above, we can formalize the notion of an unordered list as the free commutative monoid and the idea of a list as the free monoid. We can then study sorting as a relationship between these two structures under the framework of universal algebra [Birkhoff 1935]. However, to begin, we would need to explain what an algebra is, what exactly is the definition of "free".

3.1 Algebra

First we define very generally what an algebra is.

Definition 3.1.1. link *Every algebra has a signature σ, which is a (dependent) tuple of:*

- *a set of function symbols op : Set,*
- *and an arity function $ar : op \to \mathsf{Set}$.*

Here, op gives us a set of names to refer to the operations we can do in an algebra, and the arity function ar specifies the arity of the operation, using the cardinality of the output set as the arity. In other definitions, the arity function may be defined as $ar : op \to \mathbb{N}$, which specifies the arity of an operation op by the output natural number. However, for generality, we decide to use hSet and cardinality for arity instead, so that we can have operations of infinite arity.

Example 3.1.2. link *We can define the signature of monoid as follow: the algebra* Mon *has operation symbols $\{e, \bullet\}$, referring to the identity operation and the multiplication operation. The arity function is defined as: $ar := \{e \mapsto \mathsf{Fin}_0, \bullet \mapsto \mathsf{Fin}_2\}$.*

We use X^Y to denote a vector of X from now on, with the dimension of the vector given by the cardinality of Y. We now can define the type $F_\sigma(X) := \Sigma_{(f:op)} X^{ar(f)}$, which would be an endofunctor on Set. We refer to this endofunctor as the signature endofunctor. In plain English, $F_\sigma(X)$ is a tuple of an operation symbol op and a vector of X, where the dimension of the vector is the arity of op.

Definition 3.1.3. link *A σ-structure $\mathfrak{X}$ is an F_σ-algebra, given by a tuple of:*

- *a carrier set X : hSet,*
- *and an interpretation function $\alpha_X : F_\sigma(X) \to X$.*

In the rest of the dissertation, we use a normal alphabet (e.g. X) to denote the carrier set, and the alphabet in Fraktur font (e.g. $\mathfrak{X}$) to denote an algebra based on the carrier set.

Example 3.1.4. link *($\mathbb{N}$, +) is a* Mon*-algebra, with the carrier set given by $\mathbb{N}$ and its interpretation function given by:*

$$\alpha_X(e) = 0$$
$$\alpha_X(\bullet, [a, b]) = a + b$$

Definition 3.1.5. link *A F_σ-algebra homomorphism $h : X \to Y$ is a function such that:*

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

Informally, this means that for any n-ary function symbol $(f, [x_1, \ldots, x_n]) : F_\sigma(X)$,

$$h(\alpha_X(f, [a_1, \ldots, a_n])) = \alpha_Y(f, [h(x_1), \ldots, h(x_n)])$$

F_σ-algebras and their morphisms form a category F_σ-Alg. For brevity, we also refer to this category as σ-Alg. The category is given by the identity homomorphism and composition of homomorphisms.

3.2 Free Algebra

The category σ-Alg admits a forgetful functor $\sigma - \mathsf{Alg} \to \mathsf{Set}$, which forgets all the algebraic structure and only retains the underlying carrier set. We use U to denote the forgetful functor, and under our notation $U(\mathfrak{X})$ is simply X.

Definition 3.2.1. link *The free σ-algebra $\mathfrak{F}$ is simply defined as the left adjoint to U. More concretely, it is given by:*

- *a type constructor $F : \mathsf{Set} \to \mathsf{Set}$,*
- *a universal generators map $\eta_X : X \to F(X)$, such that*
- *for any σ-algebra $\mathfrak{Y}$, post-composition with η_X is an equivalence:*

$$\begin{tikzcd}
\mathfrak{F}(X) \arrow[d, dotted, "f"] & \mapsto & X \arrow[r, "\eta_X"] \arrow[rd, "f \circ \eta_X"'] & F(X) \arrow[d, dotted, "f"] \\
\mathfrak{Y} & & & Y
\end{tikzcd}$$

More concretely, let f be a morphism from $\mathfrak{F}(X) \to \mathfrak{Y}$. f would be a homomorphism since $\mathfrak{F}(X)$ and $\mathfrak{Y}$ are objects in the category σ-Alg. We use $- \circ \eta_X : (\mathfrak{F}(X) \to \mathfrak{Y}) \to (X \to Y)$ to denote the act of turning a homomorphism to a set function by post-composition with η_X. The universal property of free algebras state that $- \circ \eta_X$ is a equivalence, i.e. there is a one-to-one mapping from homomorphisms $\mathfrak{F}(X) \to \mathfrak{Y}$ to set functions $X \to Y$. Naturally, there would be an inverse mapping which maps set functions $X \to Y$ to homomorphisms $\mathfrak{F}(X) \to \mathfrak{Y}$. We refer to this as the extension operation $(-)^\sharp : (X \to Y) \to (\mathfrak{F}(X) \to \mathfrak{Y})$, and we use $f^\sharp$ to denote a homomorphism given by a set function $f : X \to Y$.

By definition, if $F : \mathsf{Set} \to \mathsf{Set}$ satisfies the universal property of free σ-algebra, then $\mathfrak{F}$ would be the free σ-algebra. We use "the" to refer to F since F is unique up to isomorphism.

Lemma 1. link *Any two free σ-algebra on the same set are isomorphic.*

Proof. Let $E(X)$ and $F(X)$ be the carrier sets of two free σ-algebras. This gives us $\eta_1 : X \to E(X)$ and $\eta_2 : X \to F(X)$. We can extend them to ${\eta_1}^\sharp : \mathfrak{F}(X) \to \mathfrak{E}(X)$ and ${\eta_2}^\sharp : \mathfrak{E}(X) \to \mathfrak{F}(X)$. By uniqueness, ${\eta_1}^\sharp$ and ${\eta_2}^\sharp$ must be the identity homomorphism, therefore $E(X)$ and $F(X)$ are isomorphic. □

3.2.1 Construction

We can construct the carrier set of the free σ-algebra $\mathfrak{F}(X)$ as an inductive type of trees $Tr_\sigma(V)$.

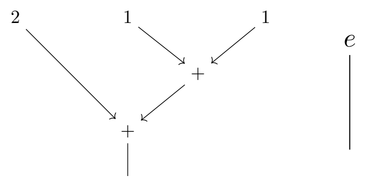
Definition 3.2.2. link *The type $Tr_\sigma(V)$ is given by:*

```
data Tree (V : 𝒰) : 𝒰 where
    leaf : V → Tree V
    node : F_σ(Tree V) → Tree V
```

The constructors $leaf$ and $node$ correspond to the generator map and the interpretation function. Concretely, $Tr_\sigma(V)$ can be thought of as an abstract syntax tree denoting operations done an elements of V.

Given a σ-algebra $\mathfrak{X}$, composing a homomorphism $f : \mathfrak{F}(V) \to \mathfrak{X}$ with $node$ with turn it into a function $V \to X$. We can define an inverse to this operation by lifting a set function $f : V \to X$ to a homomorphism $f^\sharp : \mathfrak{F}(V) \to \mathfrak{X}$ as follow: we first apply f on every leaves of the tree, giving us $Tr_\sigma(X)$, and evaluate the tree according to the operations in the nodes, giving us a X as the result.

Example 3.2.3. *A tree for σ_{Mon} with the carrier set $\mathbb{N}$ would look like:*



3.3 Equation

We have shown how to define the operations of an algebra, but we have not shown how to define the equations of an algebra. For example, a monoid should satisfy the unit laws and associative law. We now explain how we can define equations.

Definition 3.3.1. link *Every algebra also has an equation signature ϵ, which is a (dependent) tuple of:*

- *a set of equation symbols eq : Set,*
- *and an arity of free variables $fv : eq \to \mathsf{Set}$.*

Here, eq gives us a set of names to refer to an equation which the algebra must satisfy, and the arity function fv specifies the number of free variables in the equation, using the cardinality of the output set as the arity. Once again, other definitions might opt to define this as $fv : eq \to \mathbb{N}$, but we opted to use hSet and cardinality instead to allow for infinitary equations.

Example 3.3.2. link *We can define the equation signature of monoid as follow: the algebra* Mon *has equation symbols $\{\mathsf{unitl}, \mathsf{unitr}, \mathsf{assocr}\}$, referring to the unit left law, the unit right law, and the associativity law. The arity function is defined as: $fv : \{\mathsf{unitl} \mapsto \mathsf{Fin}_1, \mathsf{unitr} \mapsto \mathsf{Fin}_1, \mathsf{assocr} \mapsto \mathsf{Fin}_3\}$.*

We can the define a system of equations (or equational theory T_ϵ) by a pair of trees on the set of free variables, representing the left hand side and right hand side of the equations.

Definition 3.3.3. link *A system of equations T_ϵ is defined as $l, r : (e : eq) \to Tr_\sigma(fv(e))$.*

We say a σ-structure $\mathfrak{X}$ satisfies T iff for all $e : eq$ and $\rho : fv(e) \to X$, $\rho^\sharp(l(e)) = \rho^\sharp(r(e))$. Concretely, this means that $\rho : fv(e) \to X$ assigns free variables in equations to elements of X, and $\rho^\sharp : \mathfrak{F}(fv(e)) \to \mathfrak{X}$ evaluates a tree given an assignment of free variables. Because $\rho^\sharp$ is a homomorphism, it must evaluate correctly according to the algebraic structure of F_σ.

Example 3.3.4. *link To show (ℕ, 0, +) satisfies monoidal laws, we need to prove the following properties:*

$$\begin{aligned}\mathsf{unitl} &: \forall(\rho : \mathbb{N}^{\mathsf{Fin}_1}).\, \rho(0) + 0 = \rho(0)\\ \mathsf{unitr} &: \forall(\rho : \mathbb{N}^{\mathsf{Fin}_1}).\, 0 + \rho(0) = \rho(0)\\ \mathsf{assocr} &: \forall(\rho : \mathbb{N}^{\mathsf{Fin}_3}).\, (\rho(0) + \rho(1)) + \rho(2) = \rho(0) + (\rho(1) + \rho(2))\end{aligned}$$

3.4 Construction

To show that a construction $F(X)$ is a free σ-algebra, we first have to show it is a σ-algebra by defining how the operations of σ would compute on $F(X)$. For example, identity operation for $\mathsf{List}(X)$ is given by a constant function $\lambda().\, []$., and the $\bullet$ operation would be given by the concatenation function. We then have to show that it is a free σ-algebra by defining what the universal map $\eta : X \to F(X)$ would be, and define an extension operation $(-)^\sharp : (X \to Y) \to (\mathfrak{F}(X) \to \mathfrak{F}(Y))$ would be. Finally, we have to show that $(-)^\sharp$ is the inverse to $- \circ \eta$, therefore $F(X)$ does satisfy the universal property of free algebra.

We note that to actually show $Tr_\sigma(X)$ is a free σ-algebra, we need to quotient $Tr_\sigma(X)$ by the equations. To do this for infinitary operations would require axiom of choice [Blass 1983], therefore we can not give a general construction of free algebra as a quotient without assuming non-constructive principles. Alternatively, we can give a general construction as a HIT [Univalent Foundations Program 2013], however doing this in Cubical Agda led to termination checking issues. After a week on trying to give a general construction, we decided to move on to other issues instead, therefore we do not give a general construction in the formalization.

In the next sections, we give constructions of free monoids and free commutative monoids, and prove these constructions are correct under this framework.

4 Construction 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 Lemma 1, these constructions become equivalent (hence equal, by univalence) as types (and as monoids). By showing that these constructions satisfy the universal property of free monoids, we obtain the universal extension (fold) operation. Using fold allows us to structure our programs in an elegant way, and also exploit the properties of homomorphisms, which is used in Chapter 6.

4.1 Lists

Cons-lists are simple inductive data types, well-known to functional programmers, and are the most common representation of free monoids in programming languages. That lists are free monoids was first observed in category theory [Dubuc 1974], and in Kelly's notion of algebraically-free monoids [Kelly 1980]. As an inductive type, lists are generated by two (point) constructors: `[]` and `::` (cons).

Definition 4.1.1 (Lists)**.** *The type of lists on a type A is given by:*

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

An example of a List would be $3 :: 1 :: 2 :: []$, or simply, $[3, 1, 2]$. The (universal) generators map is the singleton: $\eta_A(a) = [a] =$ `a :: []`, the identity element is the empty list `[]`, and the monoid operation $+\!\!+$ is given by the concatenation operation.

Definition 4.1.2 (Concatenation)**.** *We define the concatenation operation* $+\!\!+ : \mathsf{List}(A) \to \mathsf{List}(A) \to \mathsf{List}(A)$*, 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 4.1.3. *link For any monoid $\mathfrak{X}$, and given a map $f : A \to X$, we define the extension $f^\sharp : \mathsf{List}(A) \to \mathfrak{X}$ by recursion on the list:*

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

Proposition 1. $(-)^\sharp$ *lifts a function $f : A \to X$ to a monoid homomorphism $f^\sharp : \mathsf{List}(A) \to \mathfrak{X}$.*

Proof sketch. The proof is formalized here. □

Proposition 2 (Universal property for List)**.** *$(\mathsf{List}(A), \eta_A)$ is the free monoid on A.*

Proof sketch. The proof is formalized here. We need to show that $(-)^\sharp$ is an inverse to $(-) \circ \eta_A$. It is trivial to see $f^\sharp \circ \eta_A = f$ for all set functions $f : A \to X$. To show $(f \circ \eta_A)^\sharp = f$ for all monoid homomorphisms $f : \mathsf{List}(A) \to \mathfrak{X}$, we need to show $\forall xs. (f \circ \eta_A)^\sharp(xs) = f(xs)$, which we can prove by induction on xs. $\square$

4.2 Array

An alternate 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 is the more well-known notion in category theory [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 $\mathsf{Fin}_n \to A$, 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 4.2.1. *link The type of* Array *on a type A is given by:*

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

An example Array is $(3, \lambda\{0 \mapsto 3, 1 \mapsto 1, 2 \mapsto 2\})$, which 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 given by array concatenation.

Lemma 2. *link Any zero-length array $(0, f)$ is equal to the identity element.*

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

Definition 4.2.2. *link We define the concatenation operation $+\!\!+ : \mathsf{Array}(A) \to \mathsf{Array}(A) \to \mathsf{Array}(A)$, and the combine operation $\oplus : (\mathsf{Fin}_n \to A) \to (\mathsf{Fin}_m \to A) \to (\mathsf{Fin}_{n+m} \to A)$.*

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

Proposition 3. *$(\mathsf{Array}(A), +\!\!+)$ is a monoid.*

Proof sketch. The proof is formalized here. We can prove $(\mathsf{Array}(A), +\!\!+)$ satisfies the unit laws by showing $(\lambda\{\} \oplus f)(k) = (f \oplus \lambda\{\})(k) = f(k)$ for all $k : \mathsf{Fin}_n$, by proving that it is impossible for $k < 0$ and $k \geq n$, therefore $(\lambda\{\} \oplus f)(k)$ and $(f \oplus \lambda\{\})(k)$ must reduce to $f(k)$. For associativity, we need to show $((f \oplus g) \oplus h)(k) = (f \oplus (g \oplus h))(k)$ for all $k : \mathsf{Fin}_{n+m+o}$. We show this by first case matching on $k < n + m$, then on $k < n$. $\square$

Lemma 3 (Array cons)**.** *Any array $(S(n), f)$ is equal to $\eta_A(f(0)) +\!\!+ (n, f \circ S)$.*

Lemma 4 (Array split)**.** *For any array $(S(n), f)$ and (m, g), we have*

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

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 4.2.3. *link Given a monoid $\mathfrak{X}$, and a map $f : A \to X$, we define $f^\sharp : \mathsf{Array}(A) \to X$, 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 4. *$(-)^\sharp$ lifts a function $f : A \to X$ to a monoid homomorphism $f^\sharp : \mathsf{Array}(A) \to \mathfrak{X}$.*

Proposition 5 (Universal property for Array)**.** *$(\mathsf{Array}(A), \eta_A)$ is the free monoid on A.*

Proof sketch. The proof is formalized here. We need to show that $(-)^\sharp$ is an inverse to $(-) \circ \eta_A$. It is trivial to see $f^\sharp \circ \eta_A = f$ for all set functions $f : A \to X$. To show $(f \circ \eta_A)^\sharp = f$ for all monoid homomorphism $f : \mathsf{Array}(A) \to \mathfrak{X}$, we need to show $\forall xs. (f \circ \eta_A)^\sharp(xs) = f(xs)$. Lemma 3 and Lemma 4 allow us to perform 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 proved for List. □

4.2.1 Remark on representation

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. This proof is also available in our formalization.

5 Construction of Free Commutative Monoids

The next step for us is to add commutativity – extending our constructions of free monoids to free commutative 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 from unordered lists to ordered lists. which is later in Section 7.1.

It is known that finite multisets are commutative monoids (in fact, free commutative monoids), under the operation of multiset union: $xs \cup ys = ys \cup xs$. Its order is "forgotten" in the sense that it doesn't matter how two multisets are union-ed together, or in more concrete terms, for example, $\{a, a, b, c\} = \{b, a, c, a\}$ are equal as finite multisets This is unlike free monoids, using List for example, where $[a, a, b, c] \neq [b, a, c, a]$.

5.1 Free monoids with a quotient

Instead of constructing free commutative monoids directly, the first construction we study is to take a free monoid and impose commutativity on it. We do this generally, by giving a construction of free commutative monoid as a quotient of *any* free monoid. Specific instances of this construction are given in Section 5.2 and Section 5.4.

From the universal algebraic perspective developed in Chapter 3, we can ask what it means to extend the equational theory of a given algebraic signature – or how to construct a free commutative monoid as a quotient of a free monoid. 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 relation* $\approx$, that turns it into a free commutative monoid. This is exactly the specification of a *permutation relation*!

Definition 5.1.1. *link* *A relation $\approx$ on free monoid 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 $q : F(A) \to F(A)/\approx$ be the quotient map (inclusion into the quotient). The generators map is given by $q \circ \eta_A$, the identity element is $q(e)$, and the $+\!\!+$ operation can also be lifted to the quotient by congruence.

Proposition 6. *link* $(\mathfrak{F}(A)/\approx, +\!\!+, q(e))$ *is a commutative monoid.*

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

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 5.1.2. *link 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 $(-)^\sharp$.*

Using the correct specification of a permutation relation, we can prove that $(\mathfrak{F}(A)/\approx)$ gives the free commutative monoid on A.

Lemma 5. *For all $f : A \to X, x : F(A)$, $f^\sharp(q(x))$ reduces to $\widehat{f}(x)$.*

Proposition 7. *link $(\mathfrak{F}(A)/\approx, \eta_A : A \xrightarrow{q} \mathfrak{F}(A) \to \mathfrak{F}(A)/\approx)$ is the free commutative 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 $f : F(A)/\approx\ \to \mathfrak{X}$ and $x : F(A)/\approx$, $(f \circ \eta_A)^\sharp(x) = f(x)$. To prove this it is suffice to show for all $x : F(A)$, $(f \circ \eta_A)^\sharp(q(x)) = f(q(x))$. $(f \circ \eta_A)^\sharp(q(x))$ reduces to $(f \circ \widehat{q \circ \mu_A})(x)$. Note that both f and q are homomorphism, therefore $f \circ q$ is a homomorphism. By universal property of F we get $(f \circ \widehat{q \circ \mu_A})(x) = (f \circ q)(x)$, 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 $F(A)/\approx\ \to \mathfrak{X}$ to set functions $A \to X$, and its inverse is given by $(-)^\sharp$, which maps set functions $A \to X$ to commutative monoid homomorphisms $F(A)/\approx\ \to \mathfrak{X}$. Therefore, $F(A)/\approx$ is indeed the free commutative monoid on A. □

5.2 Lists quotiented by permutation

A specific instance of this construction would be List quotiented with permutation to get commutativity. This construction is also considered in [Joram and Veltri 2023], who gave a proof that PList is equivalent to the free commutative monoid as a HIT. We give a direct proof of its universal property using our generalistaion.

The permutation relation on lists is as follows, which swaps any two adjacent elements in the middle of the list. This is only one example of such a relation, of course, there are many such in the literature.

Definition 5.2.1. *link*

```
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
```

We have already given a proof of List being the free monoid in Section 4.1. By Section 5.1 it suffices to show Perm satisfies the axioms of permutation relation to show PList is the free commutative monoid.

It is trivial to see how Perm satisfies reflexivity, symmetry, transitivity. We can also prove Perm is congruent wrt to $+\!\!+$ inductively. Commutativity is proven similarly, like the proof for SList in Theorem 5.3.2.

Proposition 8. *link Let X be the carrier set of a commutative monoid $\mathfrak{X}$. For all $f : A \to X$, $x, y : A$ and $xs, ys : \mathsf{PList}(A)$, $f^\sharp(xs +\!\!+ x :: y :: ys) = f^\sharp(xs +\!\!+ y :: x :: ys)$.*

With this we can prove 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 $f^\sharp$. $\square$

5.2.1 Remark on representation

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.

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 8, $f^\sharp((z :: zs) +\!\!+ x :: y :: ys) = f^\sharp([z]) \bullet f^\sharp(zs +\!\!+ x :: y :: ys)$, 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.

5.3 Swap-List

Informally, quotients generate too many points, then quotient 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-list taken from [Choudhury and Fiore 2023], where they have also given a proof of its universal property. We include this presentation for completeness, which is also a good presentation in some of our proofs.

```
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
```

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 5.3.1. *link 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.

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

[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 6. *link For all* $x : A$, $xs : \mathsf{SList}(A)$, $x :: xs = xs +\!\!+ [x]$.

Proof. We can prove this by induction on xs. For $xs = []$ this is trivial. For $xs = y :: ys$, 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 5.3.2. *link For all* $xs, ys : \mathsf{SList}(A)$, $xs +\!\!+ ys = ys +\!\!+ xs$.

Proof. By induction on xs we can iteratively apply Lemma 6 to move all elements of xs to after ys. This would move ys to the head and xs to the end, thereby proving $xs +\!\!+ ys = ys +\!\!+ xs$. ☐

5.3.1 Remark on representation

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 Section 5.2.1.

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.

5.4 Bag

Alternatively, we can also quotient Array with 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 5.4.1. *link*

```
_≈_ : 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 a permutation or an automorphism on the indices.

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

Proposition 9. *link $\approx$ 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. ☐

Proposition 10. *link $\approx$ 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 can construct τ as follow:

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

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

Figure 5.1: Operation of τ

☐

Proposition 11. *link $\approx$ 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 et al. 2022] to define ϕ:

$$\phi := \mathsf{Fin_{n+m}} \xrightarrow{\sim} \mathsf{Fin_n} + \mathsf{Fin_m} \xrightarrow{\mathsf{swap}_+} \mathsf{Fin_m} + \mathsf{Fin_n} \xrightarrow{\sim} \mathsf{Fin_{m+n}}$$

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

Figure 5.2: Operation of ϕ

□

Proposition 12. *link $\approx$ 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 et al. 2022] for studying permutation codes. This proof was particularly hard to prove and it took two weeks to prove this. The main insight comes from the proof of isomorphism between Bag and PList in [Choudhury and Fiore 2023], where the proof is split into two cases: one where $\phi(0) = 0$ and one where $\phi(0) = S(n)$ for some n. This led to the realization that the proof can be completed inductively if $\phi(0) = 0$, and all that remains is to bridge the proof from $\phi(0) = S(n)$ to $\phi(0) = 0$.

Lemma 7. *link Given $\phi \colon \mathsf{Fin_{S(n)}} \xrightarrow{\sim} \mathsf{Fin_{S(n)}}$, there is a $\tau \colon \mathsf{Fin_{S(n)}} \xrightarrow{\sim} \mathsf{Fin_{S(n)}}$ 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 can construct τ as follow:

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

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

Figure 5.3: Operation of τ constructed from ϕ

It is trivial to prove $f^\sharp(S(n), i \circ \phi) = f^\sharp(S(n), i \circ \tau)$ 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 8. *Given $\tau \colon \mathsf{Fin}_{\mathsf{S(n)}} \xrightarrow{\sim} \mathsf{Fin}_{\mathsf{S(n)}}$ where $\tau(0) = 0$, there is a $\psi : \mathsf{Fin}_{\mathsf{n}} \xrightarrow{\sim} \mathsf{Fin}_{\mathsf{n}}$ such that $\tau \circ S = S \circ \psi$.*

Proof. link We can construct ψ by $\psi(x) = \tau(S(x)) - 1$. 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 et al. 2022].

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

Figure 5.4: *Operation of ψ constructed from τ*

□

We now prove our main theorem.

Theorem 5.4.2. *link For all $\phi \colon \mathsf{Fin}_{\mathsf{n}} \xrightarrow{\sim} \mathsf{Fin}_{\mathsf{n}}$, $f^\sharp(n, i) = f^\sharp(n, i \circ \phi)$.*

Proof. We can prove this by induction on n.

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

$$\begin{aligned}
& f^\sharp(S(m), i \circ \phi) & \\
&= f^\sharp(S(m), i \circ \tau) && \text{by Lemma 7} \\
&= f(i(\tau(0))) \bullet f^\sharp(m, i \circ \tau \circ S) && \text{by definition of } (-)^\sharp \\
&= f(i(0)) \bullet f^\sharp(m, i \circ \tau \circ S) && \text{by construction of } \tau \\
&= f(i(0)) \bullet f^\sharp(m, i \circ S \circ \psi) && \text{by Lemma 8} \\
&= f(i(0)) \bullet f^\sharp(m, i \circ S) && \text{induction} \\
&= f^\sharp(S(m), i) && \text{by definition of } (-)^\sharp
\end{aligned}$$

□

5.4.1 Remark on representation

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 5, two lemmas Lemma 3 and Lemma 4 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 5.4.2, 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.

Also, 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 11, or perform operations on the partitions individually such as two individual permutation in Proposition 10. 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.

6 | Combinatorics

The purpose of all the hard work behind establishing the universal properties of our types of (ordered) and unordered lists is to be able to define operations on them systematically, which are mathematically sound, and to be able to reason about them – which is a common theme in the Algebra of Programming [Bird and de Moor 1997] community. Using our tools, we explore definitions of various combinatorial properties and operations for 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.

With universal properties we can structure and reason about our programs using algebraic ideas. 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. Within our framework, where *fold* (or the $(-)^\sharp$ operation) is a correct-by-construction homomorphism, we can define operations like length simply by extension, which also gives us a proof that they are homomorphisms for free! 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 Lemma 1), which is desirable especially when proving a direct equivalence between the two types turns out to be a difficult exercise in combinatorics.

To illustrate this, we give examples of some common operations that are defined in terms of the universal property.

6.1 Length

Any presentation of free monoids or free commutative monoids has a $\mathsf{length} : \mathcal{F}(A) \to \mathbb{N}$ function. $\mathbb{N}$ is not just a monoid with $(0, +)$, the $+$ operation is also commutative! We can extend the constant function $\lambda x.\, 1 : A \to \mathbb{N}$ to obtain $(\lambda x.\, 1)^\sharp : \mathcal{F}(A) \to \mathbb{N}$, which is the length homomorphism. This allows us to define length for any construction of free (commutative) monoids, and also gives us a proof of length being a homomorphism for free.

$$\begin{tikzcd}
\mathcal{F}(A) \arrow[r, "(\lambda x.\, 1)^\sharp"] & (\mathbb{N}, 0, +) \\
A \arrow[u, "\eta_A"] \arrow[ur, "\lambda x.\, 1"'] &
\end{tikzcd}$$

Figure 6.1: Definition of length *by universal property*

6.2 Membership

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. By fixing an element $x : A$, the element we want to define the membership proof for, we define $\mathcal{Y}_A(x) = \lambda y.\, x = y : A \to \mathsf{hProp}$. This is formally the Yoneda map under the "types are groupoids" correspondence, where $x : A$ is being sent to the Hom-groupoid (formed by the identity type), of type hProp. Now, the main observation is that hProp forms a (commutative) monoid under logical or: $\vee$ and false: $\bot$. Note that the proofs of monoid laws use equality, which requires the use of univalence (or at least, propositional extensionality). Using this (commutative) monoid structure, we extend $\mathcal{Y}_A(x)$ to obtain $x \in - : \mathcal{F}(A) \to \mathsf{hProp}$, which gives us the membership predicate for x, and its homomorphic properties (the colluquial here / there constructors for the de Bruijn encoding).

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 $\ell + 1$. This makes it such that the Yoneda map would not give us a monad since we are going from the category of Set of level ℓ to the category of Set of 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 [Arkor and McDermott 2023] 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 Chapter 8.

The definition membership for SList is formalized here and for List it is formalized here. While it is possible to generalize the definition of membership for any free monoid and free commutative monoid under the framework, we did not do so because we did not bother to refactor the code.

$$\begin{tikzcd}
\mathcal{F}(A) \arrow[r, "(x\in-)/\mathcal{Y}_A(x)^\sharp"] & (\mathsf{hProp}, \bot, \vee) \\
A \arrow[u, "\eta_A"] \arrow[ur, "\mathcal{Y}_A(x)"'] &
\end{tikzcd}$$

Figure 6.2: Definition of membership proof for x by universal property

6.3 Any and All predicates

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. We note that 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. That is,

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

$$\begin{tikzcd}
\mathcal{F}(A) \arrow[r, "P^\sharp"] & (\mathsf{hProp}, \top, \wedge) \\
A \arrow[u, "\eta_A"] \arrow[ur, "P"'] &
\end{tikzcd}$$

Figure 6.3: Definition of All by universal property

$$\begin{tikzcd}
\mathcal{F}(A) \arrow[r, "P^\sharp"] & (\mathsf{hProp}, \bot, \vee) \\
A \arrow[u, "\eta_A"] \arrow[ur, "P"'] &
\end{tikzcd}$$

Figure 6.4: Definition of Any by universal property

These constructions are used in the proofs of our main results in Section 7.1.

6.4 Other combinatorial properties

Free commutative monoid and free monoids also share common combinatorial properties, for example both satisfy the conical monoid property. However, more often than not, they both do not exhibit the same properties, for example:

- Generally, $\mathcal{F}(A+B)$ is the coproduct of $\mathcal{F}(A)$ and $\mathcal{F}(B)$. For (free) commutative monoids, this is equivalent to their underlying cartesian product (dual of Fox's theorem [Fox 1976]), that is, $\mathcal{M}(A+B) \simeq \mathcal{M}(A) \times \mathcal{M}(B)$. This is important in linear logic, and used in the Seely isomorphism. However, the coproduct of two (free) monoids is given by their free product, which does not have such a characterisation.
- Free commutative monoids satisfy the Riesz refinement property, which is important in the semantics of differential linear logic, however free monoids do not.

We mention these facts for completeness – these show that the addition of commutativity leads to *unordering*, which allows shuffling around the elements of free monoids (lists), leading to interesting combinatorial structure. These are explored in more detail and proved in [Choudhury and Fiore 2023].

6.5 Heads and Tails

More generally, commutativity is a way of enforcing unordering, or forgetting ordering. The universal property of free commutative monoids only allows eliminating to another commutative monoid – can we define functions from $\mathcal{M}(A)$ to a non-commutative structure? In baby steps, we will consider the very popular head function on lists.

The head : $\mathcal{F}(A) \to A$ function takes the *first* element out of $\mathcal{F}(A)$, and the rest of the list is its tail. We analyse this by cases on the length of $\mathcal{F}(A)$ (which is definable for both free monoids and free commutative monoids).

- Case 0: head cannot be defined, for example if $A = \mathbf{0}$, then $\mathcal{F}[\mathbf{0}] \simeq \mathbf{1}$, so a head function would produce an element of $\mathbf{0}$, which is impossible.
- Case 1: For a singleton, head can be defined for both $\mathcal{M}(A)$ and $\mathcal{L}(A)$. In fact, it is a equivalence between $xs : \mathcal{F}(A)$ where $\mathsf{length}(xs) = 1$, and the type A, where the inverse is given by η_A.
- Case $n \geq 2$: head can be defined for any $\mathcal{L}(A)$, of course. Using List as an example, one can simply take the first element of the list: $\mathsf{head}(x :: y :: xs) = x$. However, for $\mathcal{M}(A)$, things go wrong! Using SList as an example, by swap, head must satisfy: $\mathsf{head}(x :: y :: xs) = \mathsf{head}(y :: x :: xs)$ for any $x, y : A$ and $xs : \mathsf{SList}(A)$. Which one do we pick – x or y? Informally, this means, we somehow need an ordering on A, or a sorted list xs of A, so we can pick a "least" (or "biggest"?) element $x \in xs$ such that for any permutation ys of xs, also $\mathsf{head}(ys) = x$.

This final observation leads to our main result in Section 7.1.

7 | Total Orders and Sorting

7.1 Total orders and Sorting

Since the free commutative monoid is also a monoid, there is a canonical map (monoid homomorphism) $q : \mathcal{L}(A) \to \mathcal{M}(A)$, which is given by $\eta_A{}^\sharp$. Since $\mathcal{M}(A)$ is (equivalently), a quotient of $\mathcal{L}(A)$ by the commutativity (or permutation) relation, it is a surjection (an effective 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 is forgetting the order (that was imposed by the list).

The classical principle of Axiom of Choice says every that surjection f has a section (right-inverse) s, that is: $\forall x.\, f(s(x)) = x$. Or in informal terms, given a surjection which includes into a quotient, a section (uniformly) picks out a canonical representative for each equivalence class. This begs the question: in our construction, 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 7.1: *Relationship of $\mathcal{L}(A)$ and $\mathcal{M}(A)$*

Since the quotienting relation is a permutation relation (from Section 5.1), a section s to q would pick a canonical representative out of the equivalence class generated by permutation. 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 a sorting algorithm, and the implications of its existence is that A can be ordered, or sorted.

We now study sort functions as sections and their relationship with total orders. First, we recall the axioms of a total order $\leq$ on a set A:

- 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 $x = y$
- totality: forall x and y, we have *merely* either $x \leq y$ or $y \leq x$

In the context of this paper, we also want to make a distinction between "decidable total order" and just "total order". A decidable total order should also satisfy

- decidability: forall x and y, we have either $x \leq y$ or $\neg(x \leq y)$

This is a stronger version of the totality axiom, where with totality 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. Given this assumption on A, we can sort!

7.2 Section from Order

Proposition 13. *link Assume there is 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. We can construct such a section by any sorting algorithm. In our formalization we chose insertion sort $\mathsf{SList}(A) \to \mathsf{List}(A)$ due to its inductive properties and simple definition. However, if we want to define other sorting algorithms, e .g. merge-sort, other representations such as $\mathsf{Bag} \to \mathsf{Array}$ would be more suitable as we have explained in Section 5.4.1. It is easy to see how this proposition holds: $q(s(xs))$ orders an unordered list xs by s, and re-discards the order again by q, the act of imposing and then forgetting an order on xs simply *permutes* the elements of xs, which is the proof of $q(s(xs)) = xs$. □

7.3 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 relation that satisfies all axioms of total order, except transitivity! We use $\{x, y, \dots\}$ to denote $\eta_A(x) \bullet \eta_A(y) \bullet \dots : \mathcal{M}(A)$, and $[x, y, \dots]$ to denote $\eta_A(x) \bullet \eta_A(y) \bullet \dots : \mathcal{L}(A)$, or $x :: xs$ to denote $\eta_A(x) \bullet xs : \mathcal{L}(A)$.

Definition 7.3.1. *link Given a section s, we define:*

$$\begin{aligned} \mathsf{least}(xs) &:= \mathsf{head}(s(xs)) \\ x \preccurlyeq y &:= \mathsf{least}(\{x, y\}) = x \ . \end{aligned}$$

Conceptually, least sorts xs by s, and picks the first element of the sorted list by head. This is the main insight, using which we establish our result. First, we observe some properties of $\preccurlyeq$.

Proposition 14. *link $\preccurlyeq$ is decidable iff A has decidable equality.*

Proposition 15. *link $\preccurlyeq$ is reflexive, antisymmetric, and total.*

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

Since apparently the act of picking a canonical representative out of unordered lists must be doing sorting somehow, we operated under the assumption that it must be possible to construct a total order from any section s. However, after being stuck for two weeks, it would appear that proving transitivity of $\preccurlyeq$ is impossible. Eventually, we showed that it is indeed impossible, by considering the following section:

Proposition 16. *$\preccurlyeq$ is not necessarily transitive.*

Proof. We give a counter-example of s that would violate transitivity. We can first define a correct sort function $\mathsf{sort} : \mathsf{SList}(\mathbb{N}) \to \mathsf{List}(\mathbb{N})$ which sorts $\mathsf{SList}(\mathbb{N})$ ascendingly, and then we can use sort to construct the counter-example s.

$$s(xs) = \begin{cases} \mathsf{sort}(xs) & \text{if } \mathsf{length}(xs) \text{ is odd} \\ \mathsf{reverse}(\mathsf{sort}(xs)) & \text{otherwise} \end{cases}$$
$$s([2,3,1,4]) = [4,3,2,1]$$
$$s([2,3,1]) = [1,2,3]$$

□

As we can see, we need more constraints on s to prove $\preccurlyeq$ is transitive. These constraints lead to the axioms of sorting!

Definition 7.3.2. *Given a list $xs : \mathcal{L}(A)$, we say xs is in the image of s* $\mathsf{in\text{-}image}_s(xs)$ *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 sorted lists, therefore $\mathsf{in\text{-}image}_s(xs)$ would imply xs is a sorted (or ordered) list.

Proposition 17. *link $x \preccurlyeq y$ iff* $\mathsf{in\text{-}image}_s([x, y])$*.*

To prevent this class of counter-examples, we now introduce our first axiom of sorting:

Definition 7.3.3. *link A section s to q satisfies* image-respects-pair *iff:*

$$\forall x\, y\, xs.\ \mathsf{in\text{-}image}_s(x :: xs) \land y \in (x :: xs) \to \mathsf{in\text{-}image}_s([x, y])\ .$$

Here we use the definition of $\in$ from Section 6.2. Informally, if s were a sort function, this states if x is the smallest element of a sorted list ys, then for any element $y \in ys$, $[x, y]$ should be also a sorted list. This ensures that s behaves correctly on 2-element lists.

Proposition 18. *link If A has a total order $\leq$, insertion sort $\mathcal{M}(A) \to \mathcal{L}(A)$ defined with $\leq$ satisfies* image-respects-pair*.*

Proposition 19. *link Assuming s satisfies* image-respects-pair*, $\preccurlyeq$ is transitive.*

Proof. We use a weaker version of the axiom where $x :: xs$ is fixed to have length of 3. We are given $x \preccurlyeq y$ and $y \preccurlyeq z$, we want to show $x \preccurlyeq z$. Consider the 3-element $\{x, y, z\} : \mathcal{M}(A)$. Let u be $\mathsf{least}(\{x, y, z\})$, by Definition 7.3.3 and Proposition 17, we have $u \preccurlyeq x \land u \preccurlyeq y \land u \preccurlyeq z$. Since $u \in \{x, y, z\}$, u must be one of the element. We prove transitivity by case splitting. If $u = x$ we have $x \preccurlyeq z$. If $u = y$ we have $y \preccurlyeq x$, and since $x \preccurlyeq y$ and $y \preccurlyeq z$ by assumption, we have $x = y$ by antisymmetry and then we have $x \preccurlyeq z$ by substitution. Finally, if $u = z$, we have $z \preccurlyeq y$, and since $y \preccurlyeq z$ and $x \preccurlyeq y$ by assumption, we have $z = y$ by antisymmetry and then we have $x \preccurlyeq z$ by substitution. □

7.4 Embedding orders into sections

We have shown that a section s that satisfies image-respects-pair would imply a total order $x \preccurlyeq y := \mathsf{least}(\{x, y\}) = x$, and a total order $\leq$ would imply a section that satisfies image-respects-pair which can be constructed by insertion sort with $\leq$. Now, can we construct a full equivalence between sections $\mathcal{M}(A) \to \mathcal{L}(A)$ that satisfy image-respects-pair and total orders on A?

Proposition 20. *link Assume A has a decidable total order $\leq$, we can construct a section s that satisfies* image-respects-pair*, such that $\preccurlyeq$ constructed from s is equivalent to $\leq$.*

Proof. We define our section $s : \mathcal{M}(A) \to \mathcal{L}(A)$ satisfying image-respects-pair to be insertion sort defined on A by $\leq$. It is trivial to see $\mathsf{in\text{-}image}_s([x, y])$ iff $x \leq y$ by the definition of insertion sort. By Proposition 17 we can see $x \preccurlyeq y$ iff $x \leq y$. We now have a total order $x \preccurlyeq y$ equivalent to $x \leq y$. □

We now have one side of the isomorphism, and showed the set of decidable total order of A can be embedded into the set of sections s that satisfy image-respects-pair.

7.5 Equivalence of order and sections

To upgrade the embedding to an isomorphism, all that is left is to show we can turn a section satisfying image-respects-pair to a total order $\preccurlyeq$, and construct the same section back from $\preccurlyeq$. Unfortunately, this fails!

Proposition 21. *link Assume A is a set with different elements, i.e. $\exists x, y : A. x \neq y$, we cannot construct a full equivalence between sections that satisfy* image-respects-pair *and total orders on A.*

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

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

By Proposition 20 we can use sort to construct $\preccurlyeq$ which would be equivalent to $\leq$. However, the $\preccurlyeq$ 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 image-respects-pair maps to the same total order, there cannot be a full equivalence. □

As we can see, we need more constraints to prove a full equivalence. We introduce our second axiom of sorting:

Definition 7.5.1. *link A section s to q satisfies* image-can-induct *iff* $\forall x\, xs.\ \mathsf{in\text{-}image}_s(x :: xs) \to \mathsf{in\text{-}image}_s(xs)$.

To prove that this axiom is correct, we need to show a section s satisfying image-respects-pair and image-can-induct would be equal to insertion sort parameterized by the $\preccurlyeq$ constructed from s. To prove this, we introduce an inductive data type for a witness of sorted lists, taken from [Appel 2023].

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

Proposition 22. *link Given an order $\leq$, for any $xs, ys : \mathcal{L}(A)$, $q(xs) = q(ys) \wedge \mathsf{Sorted}_{\leq}(xs) \wedge \mathsf{Sorted}_{\leq}(ys) \to xs = ys$.*

Informally, this states if xs and ys have the same elements, i.e. they belong to the same equivalence class generated by permutations, and if they are both sorted by $\leq$, then they would be the same.

Corollary 7.5.2. *link 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 $\leq$.*

It is trivial to see how insertion sort by $\leq$ would always produce lists that satisfy $\mathsf{Sorted}_{\leq}$. Therefore, any functions that also always produce lists that satisfy $\mathsf{Sorted}_{\leq}$ would equal to insertion sort by function extensionality.

Corollary 7.5.3. *link Given a section s that satisfies* image-respects-pair *and* image-can-induct*, let $\preccurlyeq$ be the order derived from s, $\forall xs.\ \mathsf{Sorted}_{\preccurlyeq}(s(xs))$.*

Proof. For lists of length 0 and 1 this is trivial. For further cases we need to prove by induction: given a $xs : \mathcal{M}(A)$ of length ≥ 2, let $s(xs) = x :: y :: ys$. We need to show $x \preccurlyeq y \wedge \mathsf{Sorted}_{\preccurlyeq}(y :: ys)$ to construct $\mathsf{Sorted}_{\preccurlyeq}(x :: y :: ys)$. By image-respects-pair we have $x \preccurlyeq y$, and by image-can-induct we can prove inductively $\mathsf{Sorted}_{\preccurlyeq}(y :: ys)$. □

We can now prove the full equivalence:

Proposition 23. *link Assume A has a decidable total order $\leq$, then we can construct a section $t_{\leq}$ that satisfies* image-respects-pair *and* image-can-induct*, such that if the order $\preccurlyeq$ is derived from such a section s, $t_{\preccurlyeq} = s$.*

Proof. From s we can construct a decidable total order $\preccurlyeq$ since s satisfies image-respects-pair and A has decidable equality by assumption. We construct $t_{\preccurlyeq}$ as insertion sort parameterized by $\preccurlyeq$ constructed from s. By Corollary 7.5.2 and Corollary 7.5.3, $s = t_{\preccurlyeq}$. □

Proposition 24. *link Assume A has a decidable total order $\leq$, then A has decidable equality.*

Proof. Since $\leq$ is decidable, we can use it to show A has decidable equality as follow: We decide if $x \leq y$ and $y \leq x$ and perform case splitting:

- Case $x \leq y$ and $y \leq x$: by antisymmetry, $x = y$.
- Case $\neg(x \leq y)$ and $y \leq x$: if $x = y$, then $x \leq y$, leading to contradiction by $\neg(x \leq y)$. Therefore $x \neq y$.
- Case $x \leq y$ and $\neg(y \leq x)$: Similar logic as above.
- Case $\neg(x \leq y)$ and $\neg(y \leq x)$: by totality we must have either $x \leq y$ or $y \leq x$, therefore contradiction, this case would never occur.

□

We can now state our main theorem: a well-behaved section $s : \mathcal{M}(A) \to \mathcal{L}(A)$ to the canonical map $q : \mathcal{L}(A) \to \mathcal{M}(A)$ should satisfy the following two axioms, which axiomatizes a correct sorting algorithm.

Definition 7.5.4.

- image-respects-pair*: any 2-element list can be correctly sorted*
  $\forall x\ y\ xs.\ \mathsf{in\text{-}image}_s(x :: xs) \wedge y \in (x :: xs) \to \mathsf{in\text{-}image}_s(x :: y :: [])$,
- image-can-induct*: the first axiom can be coherently lifted to 2+-element list*
  $\forall x\ xs.\ \mathsf{in\text{-}image}_s(x :: xs) \to \mathsf{in\text{-}image}_s(xs)$.

Theorem 7.5.5. *link 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 $\mathsf{isDiscrete}(A)$ be a predicate which states A has decidable equality. We have the function $o2s\colon\ \mathsf{DecTotOrd}(A) \to \mathsf{Sort}(A) \times \mathsf{isDiscrete}(A)$ is an equivalence.*

Proof. $o2s(\leq)$ can be constructed by parameterizing insertion sort with $\leq$. By Proposition 24 we show that A satisfy isDiscrete. The inverse $s2o(s)$ can be constructed by Definition 7.3.1, which is proven to be a total order by Proposition 15 and Proposition 19, and decidable since we assume $\mathsf{isDiscrete}(A)$.

By Proposition 20 we show that $s2o \circ o2s = id$, and by Proposition 23 we show that $o2s \circ s2o = id$, therefore we have a full equivalence. □

7.5.1 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.

Proposition 25. *Assume a totally ordered set A. A sorting algorithm is a function* $\mathsf{sort} : \mathcal{L}(A) \to \mathsf{O}\mathcal{L}(A)$*, that turns lists into ordered lists, where* $\mathsf{O}\mathcal{L}(A)$ *is* $\sum_{(xs:\mathcal{L}(A))} \mathsf{Sorted}_{\leq}(xs)$*, such that:*

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

Proof sketch. Using our section s, we have $p : \forall xs.\ \mathsf{Sorted}_{\preccurlyeq}(s(xs))$ by Corollary 7.5.3. We set $\mathsf{sort} = (s \circ q,\ p \circ q)$, and the rest follows by calculation. □

8 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 constructions are generalized to all type-levels. The free algebra framework in the formalization is parameterized by any h-levels, but it currently only works with sets. We also note the axioms of sorting in the formalization is named differently, in-image$_s$ is is-sorted image-respects-pair is is-head-least, and image-can-induct is is-tail-sort. We give a table of the Agda file names and their corresponding sections in the paper (Cubical.Structures is shortened to CS).

Code	Description	Reference
`index`	Index of all files	N/A
`CS.Free`	Free algebra	(Section Section 3.2)
`CS.Sig`	Algebra signature	(Section Section 3.1)
`CS.Str`	Algebra structure	(Section Definition 3.1.3)
`CS.Eq`	Algebra equations	(Section Section 3.3)
`CS.Tree`	Tree	(Section Section 3.4)
`CS.Set.Mon.List`	List	(Section Section 4.1)
`CS.Set.Mon.Array`	Array	(Section Section 4.2)
`CS.Set.CMon.QFreeMon`	Quotiented-free monoid	(Section Section 5.1)
`CS.Set.CMon.PList`	Quotiented-list	(Section Section 5.2)
`CS.Set.CMon.SList`	Swapped-list	(Section Section 5.3)
`CS.Set.CMon.Bag`	Bag	(Section Section 5.4)
`CS.Set.CMon.SList.Sort`	Sort and order relationship	(Section Section 7.1)

Table 8.1: *Status of formalised results*

9 Discussion

We conclude by discussing some background information regarding the project, related works, and future directions we can take.

The project was originally meant to be a continuation to the paper [Choudhury and Fiore 2023], where free commutative monoids are studied under HoTT. We meant to generalize the study of free commutative monoids on sets to groupoids, therefore giving us free symmetric monoidal categories.

We decided to rewrite the formalization done in the paper so that it would have a cleaner codebase, ideally one that would unify all the proofs related to the universal property of free algebras, therefore the project evolved into developing a framework for universal algebra, with the goal to first start from algebra on sets and eventually generalizing it to groupoids so we can study symmetric monoidal categories. To demonstrate the effectiveness of the framework, we surveyed different constructions of free monoids and free commutative monoids and formalized them under the framework. Many constructions of free commutative monoids come from [Joram and Veltri 2023], which proves their universal property by isomorphism to a known construction of free commutative monoid as a HIT. We opted to take a different route, which is to prove their universal property directly, giving us the new proof of Bag's universal property in Section 5.4, so as to compare the difficulty of proof by isomorphism and direct proofs. No definite conclusions were drawn, other than that the main difficulty comes from technical reasons, namely Agda not being able to reduce a huge Bag or Array for reasons we have highlighted in Section 5.4.1.

After having formalized different constructions of free commutative monoids, we decided to move on to generalizing the framework to groupoids, and also prove SList generalized to groupoids would be a free symmetric monoidal category. This proved to be quite difficult, due to the lack of existing literature on this topic and the amount of mathematical background knowledge involved. Out of concern that it cannot be completed within half a semester, we moved on to other goals, which was to study the definition of sorting under the framework, and to investigate its relationship to the well-ordering theorem, and equivalently, the axiom of choice.

The correctness of sorting algorithms has been studied extensively, with the most simple specification being that of [Appel 2023], which is defined for $\mathsf{List}(\mathbb{N}) \to \mathsf{List}(\mathbb{N})$, however it can be generalized to any set A with a total order $\leq$ easily. A more refined study has been done by [Hinze et al. 2012], and further refined in [Alexandru 2023], in which sorting algorithms are studied under a categorical framework. Their study not only studies the extensional correctness of sorting algorithms but also the algorithm itself, which allows them to derive sorting algorithm "for free". Our work compliment theirs in that we are not interested in the computation properties of sorting, but rather the abstract properties of sorting, independent of a given ordering, with the goal to eventually study how assuming the existence of sorting functions on sets would imply constructive taboos, namely the axiom of choice. While we could not draw the relationship of sorting and axiom of choice due to the lack of time, we still managed to find new interesting properties of sorting functions in Section 7.1, where we define order in terms of sorting, as opposed to defining sorting from orders.

9.1 Conclusion

With Cubical Agda, we have developed a framework for universal algebra which allows us to prove general properties of free algebras. We surveyed existing constructions of free monoids and free commutative monoids and formalized them under the framework, most notably giving a new proof of Bag's universal property. We also study how commutativity impose the notion of unorderedness, and how free commutative monoids can be used to formalize the notion of unordered lists. We then study the correctness of sorting functions from the lens of universal algebra, studying them as the section to the canonical map from free monoids to free commutative monoids. We arrive at a new axiomatization of sorting functions, in which sorting functions are not defined in terms of total order, but rather defined purely in terms of property of functions.

In the future, it would be interesting to build upon the work done on this dissertation to explore how our abstract properties of sorting can be generalized to other inductive types, such as binary trees, which might give new insight into the constructions of binary search trees. It would also be interesting to pursue our original goals, such as investigating the relationship between sorting and axiom of choice, and generalizing the framework to groupoids so we can study free symmetric monoid categories in HoTT.

9 Bibliography

M. Abbott, T. Altenkirch, and N. Ghani. Categories of Containers. In A. D. Gordon, editor, *Foundations of Software Science and Computation Structures*, Lecture Notes in Computer Science, pages 23–38, Berlin, Heidelberg, 2003. Springer. ISBN 978-3-540-36576-1. doi: 10.1007/3-540-36576-1_2.

G. C. C. Alexandru. Intrinsically correct sorting using bialgebraic semantics. Master's thesis, Radboud University, 2023. URL `https://www.ru.nl/icis/education/master-thesis/vm/theses-archive/`.

T. Altenkirch, T. Anberrée, and N. Li. Definable Quotients in Type Theory. 2011. URL `http://www.cs.nott.ac.uk/~psztxa/publ/defquotients.pdf`.

A. W. Appel. *Verified Functional Algorithms*, volume 3. Aug. 2023. URL `https://softwarefoundations.cis.upenn.edu/vfa-current/index.html`.

N. Arkor and D. McDermott. The formal theory of relative monads, May 2023. URL `http://arxiv.org/abs/2302.14014`. arXiv:2302.14014 [math].

R. Bird and O. de Moor. *Algebra of programming*. Prentice-Hall, Inc., USA, 1997. ISBN 978-0-13-507245-5.

G. Birkhoff. On the Structure of Abstract Algebras. *Mathematical Proceedings of the Cambridge Philosophical Society*, 31(4):433–454, Oct. 1935. ISSN 1469-8064, 0305-0041. doi: 10.1017/S0305004100013463. URL `https://www.cambridge.org/core/journals/mathematical-proceedings-of-the-cambridge-philosophical-society/article/on-the-structure-of-abstract-algebras/D142B3886A3B7A218D8DF8E6DDA2B5B1`.

A. Blass. Words, free algebras, and coequalizers. *Fundamenta Mathematicae*, 117(2):117–160, 1983. ISSN 0016-2736, 1730-6329. doi: 10.4064/fm-117-2-117-160. URL `http://www.impan.pl/get/doi/10.4064/fm-117-2-117-160`.

E. Brady. Idris 2: Quantitative Type Theory in Practice, Apr. 2021. URL `http://arxiv.org/abs/2104.00480`. arXiv:2104.00480 [cs].

V. Choudhury and M. Fiore. Free Commutative Monoids in Homotopy Type Theory. *Electronic Notes in Theoretical Informatics and Computer Science*, Volume 1 - Proceedings of...:10492, Feb. 2023. doi: 10.46298/entics.10492. URL `https://entics.episciences.org/10492`.

V. Choudhury, J. Karwowski, and A. Sabry. Symmetries in reversible programming: from symmetric rig groupoids to reversible programming languages. *Proceedings of the ACM on Programming Languages*, 6(POPL):6:1–6:32, Jan. 2022. doi: 10.1145/3498667. URL `https://doi.org/10.1145/3498667`.

C. Cohen, T. Coquand, S. Huber, and A. Mörtberg. Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom. page 34 pages, 2018. doi: 10.4230/LIPICS.TYPES.2015.5. URL `http://drops.dagstuhl.de/opus/volltexte/2018/8475/`. ZSCC: NoCitationData[s1] Artwork Size: 34 pages Medium: application/pdf Publisher: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany.

N. A. Danielsson. Bag Equivalence via a Proof-Relevant Membership Relation. In L. Beringer and A. Felty, editors, *Interactive Theorem Proving*, Lecture Notes in Computer Science, pages 149–165, Berlin, Heidelberg, 2012. Springer. ISBN 978-3-642-32347-8. doi: 10.1007/978-3-642-32347-8_11.

E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, USA, 1st edition, Sept. 1997. ISBN 978-0-13-215871-8.

E. J. Dubuc. Free monoids. *Journal of Algebra*, 29(2):208–228, May 1974. ISSN 0021-8693. doi: 10.1016/0021-8693(74)90095-7. URL https://www.sciencedirect.com/science/article/pii/0021869374900957.

T. Fox. Coalgebras and cartesian categories. *Communications in Algebra*, 4(7):665–667, Jan. 1976. ISSN 0092-7872. doi: 10.1080/00927877608822127. URL https://doi.org/10.1080/00927877608822127. Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/00927877608822127.

R. Hinze, D. W. James, T. Harper, N. Wu, and J. P. Magalhães. Sorting with bialgebras and distributive laws. In *Proceedings of the 8th ACM SIGPLAN workshop on Generic programming*, WGP '12, pages 69–80, New York, NY, USA, Sept. 2012. Association for Computing Machinery. ISBN 978-1-4503-1576-0. doi: 10.1145/2364394.2364405. URL https://dl.acm.org/doi/10.1145/2364394.2364405.

P. Joram and N. Veltri. Constructive Final Semantics of Finite Bags. In *DROPS-IDN/v2/document/10.4230/LIPIcs.ITP.2023.20*. Schloss-Dagstuhl - Leibniz Zentrum für Informatik, 2023. doi: 10.4230/LIPIcs.ITP.2023.20. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ITP.2023.20.

G. M. Kelly. 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):1–83, Aug. 1980. ISSN 1755-1633, 0004-9727. doi: 10.1017/S0004972700006353. URL https://www.cambridge.org/core/journals/bulletin-of-the-australian-mathematical-society/article/unified-treatment-of-transfinite-constructions-for-free-algebras-free-monoids-col
FE2E25E4959E4D8B4DE721718E7F55EE. Publisher: Cambridge University Press.

N. Li. Quotient types in type theory, July 2015. URL http://eprints.nottingham.ac.uk/28941/. Publisher: University of Nottingham.

J. Lurie. Higher Topos Theory, July 2008. URL http://arxiv.org/abs/math/0608040. arXiv:math/0608040.

M. Mozler. Cubical Agda: Simple Application of Fin: Lehmer Codes, 2021. URL https://github.com/agda/cubical/blob/a1d2bb38c0794f3cb00610cd6061cf9b5410518d/Cubical/Data/Fin/LehmerCode.agda. mozlerCubicalAgdaSimple.

T. Univalent Foundations Program. *Homotopy type theory: Univalent foundations of mathematics*. Univalent Foundations Program, Institute for Advanced Study, 2013. URL https://homotopytypetheory.org/book. ZSCC: 0000005.

A. Vezzosi, A. Mörtberg, and A. Abel. Cubical agda: a dependently typed programming language with univalence and higher inductive types. *Proceedings of the ACM on Programming Languages*, 3 (ICFP):87:1–87:29, July 2019. doi: 10.1145/3341691. URL https://doi.org/10.1145/3341691. ZSCC: 0000002.

V. Voevodsky. Univalent foundations project. *NSF grant application*, pages 5–17, 2010. URL https://www.math.ias.edu/~dgrayson/Voevodsky-old-files/files/files-annotated/Dropbox/For_the_web_site/univalent_foundations_project_for_website.pdf.

V. Voevodsky. Resizing Rules - their use and semantic justification, 2011. URL https://www.math.ias.edu/vladimir/sites/math.ias.edu.vladimir/files/2011_Bergen.pdf.

B. A. Yorgey. Combinatorial species and labelled structures. *Dissertations available from ProQuest*, pages 1–206, Jan. 2014. URL https://repository.upenn.edu/dissertations/AAI3668177. ZSCC: 0000013.

A Appendix

A.1 Proofs for Section 4.1 (Lists)

Proposition 1. $(-)^\sharp$ *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 $f^\sharp(xs +\!\!+\, ys) = f^\sharp(xs) \bullet f^\sharp(ys)$. We can do so by induction on xs.

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

Case $x :: xs$:

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

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

Proposition 2 (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. $\square$

A.2 Proofs for Section 4.2 (Array)

Proposition 3. $(\mathsf{Array}(A), \mathbin{+\!\!+})$ *is a monoid.*

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

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

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

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

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

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

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

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 $\mathsf{Fin}_{\mathsf{n+0}} \to A$, 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, $(n, f) \mathbin{+\!\!+} (0, \lambda\{\}) = (n, f)$.

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

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

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

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

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

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 3 (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 4 (Array split). *For any array* $(S(n), f)$ *and* (m, g)*, we have*

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

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 4. *$(-)^\sharp$ 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 $f^\sharp(xs \mathbin{+\!\!+} ys) = f^\sharp(xs) \bullet f^\sharp(ys)$. We can do so by induction on xs.

Case $(0, g)$: We have $g = \lambda\{\}$ by Lemma 2. $f^\sharp((0, \lambda\{\}) \mathbin{+\!\!+} ys) = f^\sharp(ys)$ 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, $f^\sharp((0, \lambda\{\}) \mathbin{+\!\!+} ys) = f^\sharp(0, \lambda\{\}) \bullet f^\sharp(ys)$.

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

$$\begin{aligned}
& f^\sharp((S(n), g) \mathbin{+\!\!+} (m, h)) \\
&= f^\sharp(S(n + m), g \oplus h) && \text{by definition of concatenation} \\
&= f((g \oplus h)(0)) \bullet f^\sharp(n + m, (g \oplus h) \circ S) && \text{by definition of } (-)^\sharp \\
&= f(g(0)) \bullet f^\sharp(n + m, (g \oplus h) \circ S) && \text{by definition of } \oplus\text{, and } 0 < S(n) \\
&= f(g(0)) \bullet f^\sharp((n, g \circ S) \mathbin{+\!\!+} (m, h)) && \text{by Lemma 4} \\
&= f(g(0)) \bullet (f^\sharp(n, g \circ S) \bullet f^\sharp(m, h))) && \text{by induction} \\
&= (f(g(0)) \bullet f^\sharp(n, g \circ S)) \bullet f^\sharp(m, h)) && \text{by associativity} \\
&= f^\sharp(S(n), g) \bullet f^\sharp(m, h)) && \text{by definition of } (-)^\sharp
\end{aligned}$$

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

Proposition 5 (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 2 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:

$$\begin{aligned}
& f(S(n), g) \\
&= f(\eta_A(g(0)) \mathbin{+\!\!+} (n, g \circ S)) && \text{by Lemma 3} \\
&= f(\eta_A(g(0))) \bullet f(n, g \circ S) && \text{by homomorphism properties of } f \\
&= (f \circ \eta_A)(g(0)) \bullet (f \circ \eta_A)^\sharp(n, g \circ S) && \text{by induction} \\
&= (f \circ \eta_A)^\sharp(S(n), g) && \text{by definition of } (-)^\sharp
\end{aligned}$$

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.

$\square$