A Fresh Look at Commutativity: Free Algebraic Structures via Fresh Lists

Clemens Kupke, Fredrik Nordvall Forsberg, and Sean Watters

University of Strathclyde

Abstract. We show how types of finite sets and multisets can be constructed in ordinary dependent type theory, without the need for quotient types or working with setoids, and prove that these constructions realise finite sets and multisets as free idempotent commutative monoids and free commutative monoids, respectively. Both constructions arise as generalisations of C. Coquand's data type of fresh lists, and we show how many other free structures also can be realised by other instantiations. All of our results have been formalised in Agda.

Keywords: Free algebraic structures · Dependent Type theory · Inductive data types

1 Introduction

The type of lists is one of the most elementary inductive data types. It has been studied and used extensively by computer scientists and programmers for decades. Two conceptually similar structures are those of finite sets and multisets, which can be thought of as unordered analogues to lists. However, capturing unordered structures in a data type while maintaining desirable properties such as decidable equality and the correct equational theory is challenging.

The usual approach to formalise unordered structures in mathematics is to represent them as functions (with finite support): finite sets as $X \to 2$, and finite multisets as $X \to \mathbb{N}$, respectively. However, these representations do not enjoy decidable equality, even if the underlying type X does.

The approach taken in most programming languages is to pretend — one uses a list (or another ordered structure for efficiency) internally, but hides it and any invariants behind a layer of abstraction provided by an API. However, each set or multiset can then be represented by many different lists, meaning that the equational theory might not be correct. This is a problem in a dependently typed setting, where having equality as a first-class type allows us to distinguish between different representations of the same set.

The analogous approach in dependent type theory is to encode these invariants in an equivalence relation on lists, and define finite sets and multisets as setoids of lists plus the appropriate equivalence relation [4]. However, this merely side-steps the issue; we may still have two distinct lists which represent the same finite (multi)set. Thus, we are forced to work with the equivalence relation at all times instead of the identity type.

In the setting of homotopy type theory [30] (HoTT), we can use higher inductive types (HITs) to define the identities on an inductive type simultaneously with its elements. This allows us to bridge the gap from the setoid approach to obtain a data type which enjoys both decidable equality and the right equational theory, as demonstrated by Choudhury and Fiore [9].

However, it may not always be possible to work in HoTT; thus, the main question we set out to answer in this work is whether it is possible in ordinary dependent type theory to define data types of finite sets and multisets, which:

(i) have decidable equality iff the underlying set has decidable equality; and
(ii) satisfy the equational theories of finite sets and multisets.

For the latter, we take as our success criteria the facts that the type of finite sets is the free idempotent commutative monoid [13] and that finite multisets are the free commutative monoid. Thus, we are really aiming to find data types for the free idempotent commutative monoid and free commutative monoid, which satisfy the above property (i). We accomplish this by restricting our attention to only those sets with decidable equality that can be totally ordered. We can then form a type of sorted lists over such a set. Provided we treat the existence of the ordering data carefully, this type turns out to give us exactly finite sets when the order is strict, and finite multisets when it is non-strict.

We show that our constructions satisfy universal properties, in the sense that they are left adjoints to forgetful functors — this is the standard way to state freeness in the language of category theory. However, note that the notion of freeness is with respect to e.g. totally ordered monoids, rather than all monoids. For proving the universal properties and for defining the categories involved, we need function extensionality. Nevertheless the constructions themselves work in ordinary dependent type theory.

Related Work Fresh lists, the key inductive data type of this work, were first introduced by C. Coquand to represent contexts in the simply typed lambda calculus [11], and then highlighted as an example of an inductive-recursive definition by Dybjer [12]. The particular notion of fresh list discussed here is a minor variation of the version found in the Agda standard library [2], which generalises the notion of freshness to an arbitrary relation.

In Section 4 we discuss sorted lists and finite sets, both of which have been extensively investigated in the past. Sorted lists are one of the archetypal examples of a dependent type, with one particularly elegant treatment of them being given by McBride [21]. Meanwhile, Appel and Leroy [3] recently introduced canonical binary tries as an extensional representation of finite *maps*. These can be used to construct finite sets with elements from the index type (positive natural numbers for Appel and Leroy). The use of tries allows for significantly improved lookup performance compared to lists, and with more work, it is conceivable that finite sets with elements from an arbitrary but fixed first-order data type could be extensionally represented this way [16]. Our representation using sorted lists is not as efficient, but on the other hand works uniformly in the element type, as long as it is equipped with a total order.

In the setting of HoTT, there is a significant body of existing work. Choudhury and Fiore [9] give a treatment of finite multisets, showing how they can be constructed using HITs. Joram and Veltri [19] continue this thread with a treatment of the final coalgebra of the finite multiset functor. Earlier, Piceghello's PhD thesis [24] investigated coherence for symmetric monoidal groupoids, showing an equivalence between free symmetric monoidal groupoids and sorted lists. Building on this, Choudhury et al. [10] investigated the relationship between sorting algorithms and the symmetric group S_n, as part of a study of the groupoid semantics of reversible programming languages.

Contributions We make the following contributions:

– We show how finite sets and multisets can be constructed in ordinary dependent type theory, without using quotient types or working with setoids.
– We prove that, assuming function extensionality, our finite sets construction forms a free-forgetful adjunction between the category of sets equipped with an order relation, and the category of idempotent, commutative monoids equipped with an order relation. Similarly our finite multisets construction form an adjunction between sets equipped with an order relation and the category of commutative monoids equipped with an order relation.
– We show how the above constructions arise from instantiations of the data type of fresh lists, and how other instantiations give free left-regular band monoids, free reflexive partial monoids, free monoids, and free pointed sets.

All our constructions and results are formalised in the proof assistant Agda, using the `--safe` and `--cubical-compatible` flags. The development [31] builds on the Agda standard library, contains around 5,300 lines of code, and typechecks in around 35 seconds on an Intel i5-1145G7 laptop with 16 GiB of RAM. An HTML listing of the Agda code can be found at `https://seanwatters.uk/agda/fresh-lists/`. Each result also has a clickable hyperlink ⚙ to the corresponding formalised statement.

2 Preliminaries and setting

We work in the mathematical setting of Martin-Löf type theory, for example as realised by Agda [23]. We write $(x : A) \to B\,x$ for the dependent function type, and use curly braces $\{x : A\} \to B\,x$ when we wish to leave the argument x implicit. We write $a = b$ for the identity type, and $a := b$ for definitions.

We say that a type A is *propositional* if all its elements are equal, that is, if $(x, y : A) \to x = y$ is provable. A type is a *set* if its identity type is propositional. Many of the types we work with will turn out to be sets (indeed, at times we take this as a prerequisite), but we do *not* assume Streicher's Axiom K [27] at the meta level, which states that every type is a set. On the other hand, we also do not assume any features from homotopy type theory, but aim to stay compatible with it. We write Type for the universe of all types, and Set and Prop for the appropriate restrictions to sets and propositions, respectively.

3 Fresh Lists

In this section we introduce the key notion of fresh lists. As we will see later, depending on the notion of freshness, fresh lists can represent various data types such as lists consisting of repetitions of one element, or lists where all elements are distinct. For us the most important example of fresh lists will be sorted lists. We will use these in Sections 4 and 5 as representations of the free (idempotent) commutative monoid over a set equipped with an order relation.

In technical terms, the type of fresh lists is a parameterised data type similar to the type of ordinary lists, with the additional requirement that in order to adjoin a new element x to a list xs, that element x must be "fresh" with respect to all other elements already present in the list xs. For convenience, we use an inductive-inductive [22] simultaneous definition of the freshness predicate; the Agda standard library instead uses an inductive-recursive definition.

Definition 1 (⚙). *Given a type A and a binary relation $R : A \to A \to \mathsf{Type}$, we mutually inductively define a type $\mathit{FList}(A, R)$, together with a relation $\#_R : A \to \mathit{FList}(A, R) \to \mathsf{Type}$, by the following constructors:*

$$\begin{aligned} \mathit{nil} &: \mathit{FList}(A, R) \\ \mathit{cons} &: (x : A) \to (xs : \mathit{FList}(A, R)) \to x \ \#_R \ xs \to \mathit{FList}(A, R) \end{aligned}$$

$$\begin{aligned} \mathit{nil}_\# &: \{a : A\} \to a \ \#_R \ \mathit{nil} \\ \mathit{cons}_\# &: \{a : A\} \to \{x : A\} \to \{xs : \mathit{FList}(A, R)\} \to \{p : x \ \#_R \ xs\} \to \\ &\qquad\qquad R \ a \ x \to a \ \#_R \ xs \to a \ \#_R \ (\mathit{cons} \ x \ xs \ p\ ) \end{aligned}$$

For $a, x : A$, and $xs : \mathit{FList}(A, R)$, we say that a is fresh for x when we have $R \ a \ x$, and that a is fresh for xs when we have $a \ \#_R \ xs$.

Our presentation of fresh lists internalises the proof data in the cons constructor. One alternative "externalised" approach is to define the type of fresh lists as the type of pairs of an ordinary list, together with a single freshness proof for the whole list. This externalised presentation is isomorphic to ours, but we do not make further use of it in this work as we find it more convenient to enforce our invariants at the level of the constructors.

Proposition 2 (⚙). *For any $R : A \to A \to \mathsf{Type}$, we have $\mathit{FList}(A, R) \cong \Sigma(xs : \mathit{List}\ A).\mathit{is\text{-}fresh}_R \ xs$, where $\mathit{is\text{-}fresh}_R : \mathit{List}\ A \to \mathsf{Type}$ is defined by*

$$\begin{aligned} \mathit{is\text{-}fresh}_R \ \mathit{nil} &:= \top \\ \mathit{is\text{-}fresh}_R \ (\mathit{cons} \ x \ xs) &:= (\mathit{All} \ (R \ x) \ xs) \ \times \ (\mathit{is\text{-}fresh}_R \ xs) \end{aligned} \qquad \square$$

The definition of the type of fresh lists makes no explicit assumptions about the properties of the relation R. Note in particular that $R \ x \ y$ may or may not be propositional. However, in practice, we would like to have that two fresh lists are equal if and only if their heads and tails are equal. For this, we need to require $x \ \#_R \ xs$ to be propositional for all $x : A$ and $xs : \mathsf{FList}(A, R)$. This is the case exactly when $R \ x \ y$ is propositional for all $x, y : A$.

Proposition 3 (⚙). *Let $R : A \to A \to \mathsf{Type}$. The type $R\,x\,y$ is propositional for all $x, y : A$ if and only if $x \#_R xs$ is propositional for all $x : A$ and $xs : \mathsf{FList}(A, R)$.*

Proof. If R is propositional, then any two $p, q : x \#_R xs$ are equal by induction over p and q. In the other direction, if $p, q : R\,x\,y$, then $\mathsf{cons}_\#\ p\ \mathsf{nil}_\#, \mathsf{cons}_\#\ q\ \mathsf{nil}_\# : x \#_R [y]$, hence $p = q$ by assumption, and injectivity of $\mathsf{cons}_\#$. □

This gives us the expected characterisation of equality of fresh lists:

Corollary 4 (⚙). *Assume R is propositional. We have $\mathsf{cons}\ x\ xs\ p = \mathsf{cons}\ y\ ys\ q$ for any freshness proofs p and q if and only if $x = y$ and $xs = ys$. In particular, if A has decidable equality, then so does $\mathsf{FList}(A, R)$.* □

The following lemma tells us that when the freshness relation R is transitive, then $a \#_R xs$ can be established by a single proof that a is related to the head of xs. It follows by a straightforward induction on xs.

Lemma 5 (⚙). *If R is transitive, then for any $a, x : A$ and $xs : \mathsf{FList}(A, R)$, if $R\,a\,x$ and $x \#_R xs$ then $a \#_R xs$.* □

We next define the standard $\mathsf{Any}\ P$ predicate on fresh lists, which holds if the predicate P is satisfied by some element of the list.

Definition 6 (⚙). *Let $P : A \to \mathsf{Type}$. The family $\mathsf{Any}\ P : \mathsf{FList}(A, R) \to \mathsf{Type}$ is defined inductively by the following constructors:*

$$\begin{aligned}
\mathit{here} &: \{x : A\}\{xs : \mathsf{FList}(A,R)\}\{p : x \#_R xs\} \to P\ x \to \mathsf{Any}\ P\ (\mathsf{cons}\ x\ xs\ p)\\
\mathit{there} &: \{x : A\}\{xs : \mathsf{FList}(A,R)\}\{p : x \#_R xs\} \to \mathsf{Any}\ P\ xs \to \mathsf{Any}\ P\ (\mathsf{cons}\ x\ xs\ p)
\end{aligned}$$

Using this construction, we can now define the membership relation $\in$ on fresh lists, i.e., the type of proofs $x \in xs$ that some element of xs is equal to x.

Definition 7 (⚙). *For $x : A$ and $xs : \mathsf{FList}(A, R)$, let*

$$x \in xs := \mathsf{Any}\ (\lambda(a : A).\, x = a)\ xs\ .$$

The following lemma relates freshness and the membership relation: a is fresh for xs if and only if a is related to every element in xs.

Lemma 8 (⚙). *Let $a : A$ and $xs : \mathsf{FList}(A, R)$. We have $a \#_R xs$ if and only if $R\,a\,b$ holds for every $b : A$ such that $b \in xs$.* □

Although the freshness proofs are essential when *building* a list, if we want to do recursion on a given list, we frequently only care about the elements, not the proofs (regardless of whether the freshness relation is propositional or not). As such, we can define right fold in the same manner as for ordinary lists, and show that it is the universal way to define functions which ignore freshness proofs.

Proposition 9 ( ). *For types X and Y, there is a function*

$$\mathit{foldr} : (X \to Y \to Y) \to Y \to \mathit{FList}(X, R) \to Y$$

satisfying $\mathit{foldr}\ f\ e\ \mathit{nil} = e$ and $\mathit{foldr}\ f\ e\ (\mathit{cons}\ x\ xs\ p) = f\ x\ (\mathit{foldr}\ f\ e\ xs)$, and foldr is universal in the following sense: For all functions $h : \mathit{FList}(X, R) \to Y$, if there is $e : Y$ and $f : X \to Y \to Y$ such that $h\ \mathit{nil} = e$ and $h\ (\mathit{cons}\ x\ xs\ p) = f\ x\ (h\ xs)$, then $h\ xs = \mathit{foldr}\ f\ e\ xs$ for all $xs : \mathit{FList}(X, R)$. □

The proof is identical to the analogous one for ordinary lists [17].

4 Free Idempotent Commutative Monoids via Sorted Lists

The important mathematical concept of a (finite) set is also a useful abstract data structure for programmers. In circumstances where we are only concerned with whether a particular element is present or not, it is advantageous to represent data in an unordered form. However, the details of exactly how to do this in a programming context are not straightforward. Inductive data types such as lists and trees, for example, are inherently ordered.

In this section, we unify the two notions of finite sets and sorted lists. We instantiate fresh lists with a strict total order as the freshness relation, giving a data type for sorted lists which cannot contain duplicates, and use this as our representation of finite sets. The key idea is that instead of working with ordinary lists quotiented by permutations (as Choudhury and Fiore [9] do), we force every collection of elements to have exactly one permissible permutation via our lists being sorted-by-construction. As a direct consequence, this type admits an extensionality principle analogous to that of sets — two sorted lists are equal if and only if they have the same elements.

4.1 Sorted Lists

We begin by defining the type $\mathsf{SList}(A, <)$ of sorted duplicate-free lists over A as an instance of FList.

Definition 10 ( ). *Let A be a type, and $< : A \to A \to \mathsf{Prop}$ a propositional strict total order, i.e., $<$ is propositional, transitive, and trichotomous: for every $x, y : A$, exactly one of $x < y$ or $x = y$ or $y < x$ holds. Then let $\mathit{SList}(A, <) := \mathit{FList}(A, <)$.*

We write $\#$ for $\#_<$, for simplicity. Note that with this exclusive-disjunction presentation of trichotomy, having a constructive witness that $<$ is trichotomous immediately implies decidable equality on A. This makes intuitive sense as we would like the question of whether an element can be appended to a list to be decidable. By Hedberg's theorem, having decidable equality also means that A is a set [15].

We now define the binary operation which merges two sorted lists together, suggestively named $\cup$, with a view towards showing that $(\mathsf{SList}(A, <),\ \cup,\ \mathsf{nil})$ is an idempotent commutative monoid. We initially define the monoid multiplication only on elements, without heed to whether any appropriate freshness proofs exist that validate the definition. We then show that such proofs do exist for all inputs.

Proposition 11 (⚙). *There is* $\cup : \mathit{SList}(A,<) \to \mathit{SList}(A,<) \to \mathit{SList}(A,<)$ *with*

$$\begin{aligned} \mathit{nil} \ \cup \ ys &:= ys \\ xs \ \cup \ \mathit{nil} &:= xs \end{aligned}$$

$$(\mathit{cons}\ x\ xs\ p) \ \cup \ (\mathit{cons}\ y\ ys\ q) := \begin{cases} \mathit{cons}\ x\ (xs \ \cup \ (\mathit{cons}\ y\ ys\ q))\ r & \text{if } x < y \\ \mathit{cons}\ x\ (xs \ \cup \ ys)\ s & \text{if } x = y \\ \mathit{cons}\ y\ ((\mathit{cons}\ x\ xs\ p) \ \cup \ ys)\ t & \text{if } x > y \end{cases}$$

for freshness proofs r, s, and t of the following types, which can be computed mutually with the definition of $\cup$:

$$\begin{aligned} r &: x \ \# \ (xs \ \cup \ (\mathit{cons}\ y\ ys\ q)) \\ s &: x \ \# \ (xs \ \cup \ ys) \\ t &: y \ \# \ ((\mathit{cons}\ x\ xs\ p) \ \cup \ ys) \end{aligned}$$

Proof. Mutually with the definition of $\cup$, we prove that for all $a : A$ and $xs, ys : \mathsf{SList}(A,<)$, if a is fresh for both xs and ys, then a is fresh for $xs \ \cup \ ys$. The freshness proofs r, s, and t required can then be constructed from p and q. The proof follows by induction on both lists. If either list is nil, then the proof is trivial. Now consider the case where we must show that a is fresh for $(\mathsf{cons}\ x\ xs\ p) \ \cup \ (\mathsf{cons}\ y\ ys\ q)$, for some $x, y : A$, $xs, ys : \mathsf{SList}(A,<)$, $p : x \ \# \ xs$, and $q : y \ \# \ ys$. By trichotomy, we have three cases to consider; either $x < y$, $x = y$, or $x > y$. If $x < y$, then we must show that $a \ \# \ \mathsf{cons}\ x\ (xs \ \cup \ (\mathsf{cons}\ y\ ys\ q))$. By assumption, $a < x$, and $a \ \# \ (xs \ \cup \ (\mathsf{cons}\ y\ ys\ q))$ by the induction hypothesis. The cases for $x = y$ and $x > y$ follow by similar arguments. □

4.2 Sorted Lists form an Idempotent Commutative Monoid

We now prove that $(\mathsf{SList}(A,<), \ \cup, \ \mathsf{nil})$ is an idempotent commutative monoid. The main tool we use for this proof is an *extensionality principle* for sorted lists, which is analogous to the axiom of extensionality for sets. In order to prove the extensionality principle, we require the following lemma. Its proof follows straightforwardly from the properties of $<$.

Lemma 12 (⚙). *Let $a, x : A$, $xs : \mathit{SList}(A,<)$, and $p : x \ \# \ xs$.*

(i) If $a < x$, then $a \notin (\mathit{cons}\ x\ xs\ p)$.
(ii) If $a \ \# \ xs$, then $a \notin xs$. □

We are now ready to prove the extensionality principle, which characterises the identity type of SList.

Theorem 13 (⚙ Extensionality Principle for SList). *Given sorted lists $xs, ys : \mathsf{SList}(A,<)$, we have $(a \in xs) \longleftrightarrow (a \in ys)$ for all $a : A$ iff $xs = ys$.*

Proof. The direction from right to left is obvious. For the other direction, we proceed by induction on both lists. The case where both are nil is trivial. The cases where one is nil and the other is cons are trivially impossible.

We focus on the case where we must show $(\mathsf{cons}\ x\ xs\ p) = (\mathsf{cons}\ y\ ys\ q)$, for some $x, y : A$, $xs, ys : \mathsf{SList}(A,<)$, $p : x \# xs$ and $q : y \# ys$. Assume $(a \in \mathsf{cons}\ x\ xs\ p) \longleftrightarrow (a \in \mathsf{cons}\ y\ ys\ q)$. By trichotomy, either $x < y$, $x > y$, or $x = y$. The former two cases are impossible by Lemma 12. Therefore, $x = y$. By Corollary 4, since $<$ is proof irrelevant, it now suffices to show $xs = ys$. By the induction hypothesis, this will be the case if $(a \in xs) \longleftrightarrow (a \in ys)$. For the forward direction, assume $u : a \in xs$. Applying $\mathsf{there}\ u$ to our initial assumption, we get $a \in (\mathsf{cons}\ y\ ys\ q)$. Either $a = y$, or $a \in ys$. The former case is impossible; if $a = y$, then $a = x$ by transitivity, so by Lemma 12, $a \notin xs$. But $a \in xs$ by assumption. Contradiction. The other direction follows the same argument. □

Using the extensionality principle, it is now not hard to prove that sorted lists form an idempotent commutative monoid.

Proposition 14 (⚙). *$(\mathsf{SList}(A,<),\ \cup,\ \mathsf{nil})$ is an idempotent commutative monoid. That is, the following equations hold for all $xs, ys, zs : \mathsf{SList}(A,<)$:*

- *unit:* $(\mathsf{nil}\ \cup\ xs) = xs = (xs\ \cup\ \mathsf{nil})$
- *associativity:* $((xs\ \cup\ ys)\ \cup\ zs) = (xs\ \cup\ (ys\ \cup\ zs))$
- *commutativity:* $(xs\ \cup\ ys) = (ys\ \cup\ xs)$
- *idempotence:* $(xs\ \cup\ xs) = xs$

Proof. The unit laws are trivial. For associativity, commutativity, and idempotence, we first prove that $a \in (xs\ \cup\ ys)$ if and only if $a \in xs$ or $a \in ys$. The equations then follow more or less directly using Theorem 13. □

4.3 A Free-Forgetful Adjunction

Since singleton lists are always sorted, they clearly give an inclusion of the underlying type A into the type of sorted lists. We might thus hope that $\mathsf{SList}(A,<)$ can be characterised by the universal property of being the smallest idempotent commutative monoid generated by A, i.e., that it is the free idempotent commutative monoid. However, in order to form the type of sorted lists over some type A, we must already have a strict total order on A. And we cannot assume that we would be able to find such an order for any set; this is a weak form of the Axiom of Choice, called the Ordering Principle (OP) (see e.g. [18, §2.3]), which implies excluded middle, as proven by Swan [28]. As such, in our constructive setting, the domain of the SList functor cannot be Set, as it lacks the required data to form sorted lists. Instead of Set, we must consider a category whose objects are linearly ordered sets (in the same sense as we have used thus far, which implies decidable equality on the elements).

We also require a forgetful functor from our category of idempotent commutative monoids into this category of linearly ordered sets, which intuitively needs to retain this ordering data — we only want to forget the monoid structure. As such, instead of the category of idempotent commutative monoids, we must consider a category of such structures equipped with their own linear orders. There are some design decisions to be made in defining these categories, regarding how much structure the morphisms ought to preserve. Specifically, we must decide whether they should be monotone with respect to the ordering data of the objects. We argue that the correct decision here is (perhaps counter-intuitively) to *not* preserve the order, a choice that we will motivate more fully in Section 4.4.

Definition 15 (⚙). *Let* STO *denote the category whose objects are strictly totally ordered types, and whose morphisms are (not necessarily monotone) functions on the underlying types. That is:*

- *Objects are pairs* $(X, <_X)$ *of a type* X *together with a propositional strict total order* $<_X \colon X \to X \to \mathsf{Prop}$.
- *Morphisms from* $(X, <_X)$ *to* $(Y, <_Y)$ *are functions* $X \to Y$.

As previously remarked, the trichotomy property of $<_X$ implies that X has decidable equality, which in turn means that X is a set, by Hedberg's theorem [15].

Definition 16 (⚙). *Let* OICMon *denote the category whose objects are strictly totally ordered idempotent commutative monoids (where the monoid multiplication does not necessarily preserve ordering), and whose morphisms are (not necessarily monotone) monoid homomorphisms. That is:*

- *Objects are 4-tuples* $(X, <_X, \cdot_X, \epsilon_X)$ *of a set* X*, a propositional strict total order* $<_X \colon X \to X \to \mathsf{Prop}$*, a binary operation* $\cdot_X : X \to X \to X$*, and an object* $\epsilon_X : X$*, such that* $(X, \cdot_X, \epsilon_X)$ *is an idempotent commutative monoid.*
- *Morphisms from* $(X, <_X, \cdot_X, \epsilon_X)$ *to* $(Y, <_Y, \cdot_Y, \epsilon_Y)$ *are functions* $f : X \to Y$ *which preserve units and multiplication.*

Since morphisms in OICMon formally carry witnesses that the underlying functions preserve unit and multiplication, this could potentially make proofs of equality between such morphisms troublesome. Thankfully, because the underlying types are sets, these troubles do not materialise, as long as we have function extensionality. This is recorded in the following lemma.

Lemma 17 (⚙). *Assuming function extensionality, two morphisms of* OICMon *are equal if and only if their underlying set functions are (pointwise) equal.* □

We must now show that as well as being idempotent commutative monoids, our sorted lists also come equipped with strict total orders. We do this by defining a lifting of orders on a type to orders on sorted lists over that type, using the lexicographic order. Note that while we require the existence of an order to have an object in OICMon, the exact choice of order does not matter; any two objects in the category with the same underlying set will be isomorphic.

Proposition 18 (⚙). *Let $<$ be a propositional strict total order on a type A. Then the lexicographic order $<_L$ on $\mathsf{SList}(A, <)$, defined inductively below, is also a propositional strict total order.*

$$\begin{aligned}
\mathit{nil}_{<_L} &: \{y : A\}\ \{ys : \mathit{SList}(A,<)\}\ \{q : y \mathbin{\#} ys\} \\
&\quad \to \mathit{nil} <_L \mathit{cons}\ y\ ys\ q \\
\mathit{here}_{<_L} &: \{x, y : A\}\ \{xs, ys : \mathit{SList}(A,<)\}\ \{p : x \mathbin{\#} xs\}\ \{q : y \mathbin{\#} ys\} \\
&\quad \to x < y \to \mathit{cons}\ x\ xs\ p <_L \mathit{cons}\ y\ ys\ q \\
\mathit{there}_{<_L} &: \{x, y : A\}\ \{xs, ys : \mathit{SList}(A,<)\}\ \{p : x \mathbin{\#} xs\}\ \{q : y \mathbin{\#} ys\} \\
&\quad \to x = y \to xs <_L ys \to \mathit{cons}\ x\ xs\ p <_L \mathit{cons}\ y\ ys\ q
\end{aligned}$$ □

We can now show that SList is a functor $\mathsf{STO} \to \mathsf{OICMon}$, with action on objects given by $\mathsf{SList}(A, <_A) := (\mathsf{SList}(A, <_A), <_L, \cup, \mathsf{nil})$. We define the action on morphisms on the underlying sets, and then show that it preserves the monoid structure, and hence is a morphism in OICMon. Our implementation of map for sorted lists is essentially insertion sort; we take a function on the underlying set, apply it to each element, and insert the result into the output list.

Definition 19 (⚙). *Given two types A and B with strict total orders $<_A : A \to A \to \mathsf{Prop}$ and $<_B : B \to B \to \mathsf{Prop}$, let:*

$$\begin{aligned}
&\mathit{map} : (A \to B) \to \mathit{SList}(A, <_A) \to \mathit{SList}(B, <_B) \\
&\mathit{map}\ f\ \mathit{nil} := \mathit{nil} \\
&\mathit{map}\ f\ (\mathit{cons}\ x\ xs\ p) := \mathit{insert}\ (f\ x)\ (\mathit{map}\ f\ xs)
\end{aligned}$$

where $\mathit{insert}\ x\ xs := (\mathit{cons}\ x\ \mathit{nil}\ \mathit{nil}_{\#})\ \cup\ xs$.

We now show that map preserves the monoid structure, and hence is a morphism in OICMon. The proof uses Theorem 13.

Lemma 20 (⚙). *For all functions $f : A \to B$ and $xs, ys : \mathit{SList}(A, <)$, we have*

$$\mathit{map}\ f\ (xs\ \cup\ ys) = (\mathit{map}\ f\ xs)\ \cup\ (\mathit{map}\ f\ ys)$$ □

Similarly, assuming function extensionality and using Lemma 17, we can show that map preserves identity and composition, and hence is a functor.

Theorem 21 (⚙). *Assuming function extensionality, $\mathit{SList} : \mathsf{STO} \to \mathsf{OICMon}$ forms a functor which is left adjoint to the forgetful functor $\mathcal{U} : \mathsf{OICMon} \to \mathsf{STO}$ defined by $\mathcal{U}(X, <, \cdot, \epsilon) := (X, <)$.*

Proof. The bijection on homsets sends a monoid morphism $f : \mathsf{SList}(A, <_A) \to (B, <, \cdot, \epsilon)$ to the function $\hat{f} : A \to B$ defined by $\hat{f} := \lambda(x : A).\ f\ (\mathsf{cons}\ x\ \mathsf{nil}\ \mathsf{nil}_{\#})$, and a function $g : A \to B$ to the monoid morphism $\check{g} : \mathsf{SList}(A, <_A) \to (B, <, \cdot, \epsilon)$ defined by $\check{g} := \mathsf{foldr}\ (\lambda(a : A)(b : B).\ (g\ a) \cdot_B b)\ \epsilon_B$. The fact that $\check{\hat{f}} = f$ follows from Proposition 9. The proofs of $\hat{\check{g}} = g$ and naturality follow by unfolding the definitions and Lemma 17 — hence the assumption of function extensionality. □

4.4 Motivating the Lack of Monotonicity

We now return to our decision to not require monotonicity for the morphisms of STO and OICMon. That we require our objects to have ordering information at all could be seen as an implementation detail; the ordering is needed to form the type of sorted lists, but thereafter we would like to treat them as finite sets.

For an illustrative example, consider the different notions of map that we obtain with and without monotonicity (recall Definition 19 for the latter). With monotonicity, we would obtain a functorial action which applies a monotone function to each element in place, such that map is also monotone. However, in practice we do not find much value in respecting whichever arbitrary order was chosen; we would rather have the freedom to lift any function to act on sorted lists, and have the implementation of map handle the details. In practice, we are mostly interested in finite sets over first order inductive data types anyway, and these can always be totally ordered. More provocatively: since we work on a computer, all of our data ought to be represented by a bit pattern in the end anyway, and by considering not necessarily monotone functions, we ensure that the particular choice of ordering derived from these bit patterns play no role.

In the same spirit, one could wonder if there is actually any difference between the categories STO and Set. After all, since morphisms are not monotone, all objects in STO with the same underlying type are actually isomorphic. The following proposition makes clear what kind of choice principle is needed in order to choose a canonical representative for these isomorphism classes. Recall that the Ordering Principle states that every set can be totally ordered: for every set X, there is a strict total order on X. This principle is weaker than the Axiom of Choice, but not provable in ZF set theory [25]; in the context of Homotopy Type Theory, Swan proved that the Ordering Principle implies Excluded Middle [28].

Proposition 22 (⚙). *The Ordering Principle holds if and only if both forgetful functors* $\mathcal{U}_{\mathsf{STO}} : \mathsf{STO} \to \mathsf{Set}$ *and* $\mathcal{U}_{\mathsf{OICMon}} : \mathsf{OICMon} \to \mathsf{ICMon}$ *are equivalences.*

Proof. If the Ordering Principle holds, then each type can be equipped with an strict total order, which gives an inverse to each forgetful functor. Conversely, an inverse to the forgetful functors equips each set with a strict total order. $\square$

Thus, in the presence of the non-constructive Ordering Principle, sorted lists are the free idempotent commutative monoid over sets. However we prefer to stay constructive and ask for more input data in the form of an order instead.

5 Free Commutative Monoids via Sorted Lists with Duplicates

Finite multisets have long been applied across computer science, particularly in database theory [6]. However their unordered nature again makes representing them in a data type challenging. We have seen that when we consider fresh lists with a strict total order as the freshness relation, we obtain a data type for sorted lists which contain no duplicate elements. If we drop the requirement that the order is irreflexive, we obtain a type $\mathsf{SListD}(A, \leqslant)$ of sorted lists where repetitions are allowed. The corresponding notion of trichotemy in this setting is totality of the order (i.e., for all x and y, either $x \leqslant y$ or $y \leqslant x$), together with decidability of both the order and the equality on the underlying type.

Definition 23 (⚙). *Let A be a type with decidable equality, and $\leqslant : A \to A \to \mathsf{Prop}$ a propositional, decidable total order. Then let $\mathit{SListD}(A, \leqslant) := \mathit{FList}(A, \leqslant)$.*

Again we write $\#$ for $\#_{\leqslant}$. Using the decidability of the order, we can now again define the merge operation on sorted lists.

Proposition 24 (⚙). *There is $\cup : \mathit{SListD}(A, \leqslant) \to \mathit{SListD}(A, \leqslant) \to \mathit{SListD}(A, \leqslant)$ with*

$$\begin{aligned} \mathit{nil} \ \cup \ ys &:= ys \\ xs \ \cup \ \mathit{nil} &:= xs \\ (\mathit{cons}\ x\ xs\ p) \ \cup \ (\mathit{cons}\ y\ ys\ q) &:= \begin{cases} \mathit{cons}\ x\ (xs \ \cup \ (\mathit{cons}\ y\ ys\ q))\ r & \text{if } x \leqslant y \\ \mathit{cons}\ y\ ((\mathit{cons}\ x\ xs\ p) \ \cup \ ys)\ s & \text{otherwise} \end{cases} \end{aligned}$$

where freshness proofs r and s with the following types exist by the same argument as in Proposition 11:

$$\begin{aligned} r &: x \# (xs \ \cup \ (\mathit{cons}\ y\ ys\ q)) \\ s &: y \# ((\mathit{cons}\ x\ xs\ p) \ \cup \ ys) \end{aligned} \qquad \square$$

Just as SList corresponds to finite sets and free idempotent commutative monoids, SListD corresponds to finite multisets and free commutative monoids. Our proof strategy follows the same structure as for Theorem 13, with one notable exception — the extensionality principle as stated for SList is not true for SListD, where for example $[a, a]$ and $[a]$ have the same elements, but with different multiplicity. Put differently: as Gylterud noted, the membership relation is prop-valued for sets, but set-valued for multisets [14, § 3.5]. As such, the extensionality principle for multisets uses isomorphism rather than logical equivalence: multisets xs and ys are equal if and only if $(a \in xs) \simeq (a \in ys)$ for every element a.

However, isomorphisms can be onerous to work with formally, and we can do better. Note that there will be a function $\mathsf{count} : \mathsf{SListD}(A, \leqslant) \to A \to \mathbb{N}$ which, given a sorted list and some element of A, returns the number of occurrences of that element in the list. We can also think of this function as converting a sorted list to its multiplicity function. The extensionality principle that we will prove is the following: two sorted list with duplicates are equal if and only if their multiplicity functions are pointwise equal. We stress that we do not need to assume function extensionality for this result.

We prove the non-trivial "if" direction in two stages: pointwise equality of multiplicity functions implies isomorphism of membership relations, which in turn implies equality of sorted lists. First, we define the count function:

Definition 25 (⚙). *Let* $\mathsf{count} : \mathsf{SListD}(A, \leqslant) \to A \to \mathbb{N}$, *where:*

$$\mathsf{count}\ \mathsf{nil}\ x := 0$$

$$\mathsf{count}\ (\mathsf{cons}\ y\ ys\ p)\ x := \begin{cases} 1 + (\mathsf{count}\ ys\ x) & \text{if } x = y \\ \mathsf{count}\ ys\ x & \text{otherwise} \end{cases}$$

We collect some basic properties of the count function.

Lemma 26 (⚙).

(i) For any $x : A$ *and* $ys : \mathsf{SListD}(A, \leqslant)$, *if* $x \notin ys$ *then* $\mathsf{count}\ ys\ x = 0$.
(ii) If for all $a : A$ *we have* $\mathsf{count}\ (\mathsf{cons}\ x\ xs\ p)\ a = \mathsf{count}\ (\mathsf{cons}\ y\ ys\ q)\ a$, *then for all* $a : A$ *also* $\mathsf{count}\ xs\ a = \mathsf{count}\ ys\ a$.

Proof. Part (i) follows by induction on ys. For (ii), by decidable equality of A, either $x = y$, or $x \neq y$, and by decidability of $\leqslant$, either $x \leqslant y$, or $y \leqslant x$. Without loss of generality, assume $x \leqslant y$.

If $x = y$, then peeling away the heads will either preserve the number of as on both sides, or decrement each count by one; in either case, the conclusion follows.

If $x \neq y$, we consider the four cases where each of x and y are either equal to a or not. The case where $x \neq a \neq y$ follows by the same argument as when $x = y$. The case where $x = a = y$ is impossible since $x \neq y$. Finally, also the case where $x = a \neq y$ (or the other way around) is impossible: we have $a = x \leqslant y$ and $y \# ys$, hence $a \notin \mathsf{cons}\ y\ ys\ q$, hence $\mathsf{count}\ (\mathsf{cons}\ y\ ys\ q)\ a = 0$ by (i). But since $a = x$, we have that $\mathsf{count}\ (\mathsf{cons}\ x\ xs\ p)\ a \geqslant 1$, contradicting the assumption. □

We are now ready to prove the first step towards the extensionality principle.

Proposition 27 (⚙). *Let* $xs, ys : \mathsf{SListD}(A, \leqslant)$. *If* $\mathsf{count}\ xs\ a = \mathsf{count}\ ys\ a$ *for all* $a : A$, *then we have isomorphisms* $(a \in xs) \cong (a \in ys)$ *for all* $a : A$.

Proof. The proof proceeds by induction on both lists. The case where both lists are nil holds trivially. Both cases where one is nil and the other is not are trivially impossible. When the lists are of the form $\mathsf{cons}\ x\ xs\ p$ and $\mathsf{cons}\ y\ ys\ q$, we can apply Lemma 26 to obtain that $\mathsf{count}\ xs\ a = \mathsf{count}\ ys\ a$ for all $a : A$. Then by the induction hypothesis, there is $f : (a \in xs) \cong (a \in ys)$. We now apply decidable equality of A to make a case distinction between $x = y$ and $x \neq y$. If $x = y$, we extend the isomorphism f by sending $\mathsf{here}\, p$ to $\mathsf{here}\, p$ and shifting the old proofs of membership by there. The other case $x \neq y$ is impossible, which we now show. By Lemma 26, we have $\mathsf{count}\ xs\ x = \mathsf{count}\ ys\ x$ for all $a : A$. Hence by instantiating the hypothesis with x, we have, since $x \neq y$,

$$\mathsf{count}\ (\mathsf{cons}\ x\ xs\ p)\ x = \mathsf{count}\ (\mathsf{cons}\ y\ ys\ q)\ x = \mathsf{count}\ ys\ x = \mathsf{count}\ xs\ x$$

but also $\mathsf{count}\ (\mathsf{cons}\ x\ xs\ p)\ x = 1 + \mathsf{count}\ xs\ x$, which is a contradiction. □

We now prove the second step: sorted lists are equal if and only if they have isomorphic membership relations. We first show that we can "peel off" the same head and still have isomorphic membership relations for the tails of the lists.

Lemma 28 (⚙). *For all $b : A$, $xs, ys : \mathsf{SListD}(A, \leqslant)$, and freshness proofs p and q, if we have an isomorphism $(a \in \mathsf{cons}\ b\ xs\ p) \cong (a \in \mathsf{cons}\ b\ ys\ q)$ for every $a : A$, then we also have an isomorphism $(a \in xs) \cong (a \in ys)$ for every $a : A$.*

Proof. Given an isomorphism $f : (a \in \mathsf{cons}\ b\ xs\ p) \to (a \in \mathsf{cons}\ b\ ys\ q)$, we construct a function $g_{xs,ys} : a \in xs \to a \in ys$, and show that $g_{ys,xs}$ is the inverse of $g_{xs,ys}$. Given $u : (a \in xs)$, we have $a \in (\mathsf{cons}\ b\ ys\ q)$, by $f\ (\mathsf{there}\ u)$. There are two possible cases: if $f\ (\mathsf{there}\ u) = \mathsf{there}\ v$ for some $v : a \in ys$, then we take $g(u) = v$. Otherwise if $f\ (\mathsf{there}\ u) = \mathsf{here}\ v$ for some $v : a = b$, then we can apply f again. If $f\ (\mathsf{here}\ v) = \mathsf{there}\ w$ for some $w : a \in ys$, then we take $g(u) = w$. If $f\ (\mathsf{here}\ v) = \mathsf{here}\ w$ for some $w : a = b$, then we can derive a contradiction: since equality on A is propositional, $v = w$, and hence $f\ (\mathsf{here}\ v) = f\ (\mathsf{there}\ u)$, and applying the inverse of f to both sides, we get $\mathsf{here}\ v = \mathsf{there}\ u$. However, different constructors of an inductive type are never equal. □

Using this lemma, we can now prove the extensionality principle for sorted lists with duplicates up to isomorphism of membership. Note that this theorem is not true for ordinary lists — it relies on the lists being sorted.

Proposition 29 (⚙). *Let $xs, ys : \mathsf{SListD}(A, \leqslant)$. If for all $a : A$ we have isomorphisms $(a \in xs) \cong (a \in ys)$, then $xs = ys$.*

Proof. By induction on xs and ys; the only non-trivial case is when the lists are of the form $\mathsf{cons}\ x\ xs\ p$ and $\mathsf{cons}\ y\ ys\ q$, in which case they are equal if $x = y$ and $xs = ys$ by Corollary 4. We have $xs = ys$ by Lemma 28 and the induction hypothesis. To prove $x = y$, note that $x \in \mathsf{cons}\ y\ ys\ q$ and $y \in \mathsf{cons}\ x\ xs\ p$ by the assumed isomorphism. Thus either $x = y$, or $x \in ys$ and $y \in xs$. In the former case, we are done, and in the latter case, since also $x \# xs$ and $y \# ys$, we then have both $x \leqslant y$ and $y \leqslant x$ by Lemma 8, so that indeed $x = y$ by antisymmetry. □

Combining Propositions 27 and 29, we get a convenient characterisation of the identity type for sorted lists with duplicates.

Theorem 30 (⚙ Extensionality Principle for SListD). *For sorted lists $xs, ys : \mathsf{SListD}(A, \leqslant)$, if $\mathsf{count}\ xs\ a = \mathsf{count}\ ys\ a$ for all $a : A$, then $xs = ys$.* □

We can now put this principle to use in order to prove that sorted lists with duplicates satisfies the axioms of a commutative monoid. This is very direct, after proving that $\mathsf{count}\ (xs\ \cup\ ys)\ a = \mathsf{count}\ xs\ a + \mathsf{count}\ ys\ a$ for all $a : A$.

Proposition 31 (⚙). *$(\mathsf{SListD}(A, \leqslant), \cup, \mathsf{nil})$ is a commutative monoid.* □

From here, we can define a category DTO of propositional decidable total orders with decidable equality, whose morphisms are not necessarily monotone functions between the underlying sets, and a category OCMon of ordered commutative monoids. By using the lexicographic order $\leqslant_L$ on sorted lists, SListD can be extended to a functorial mapping $\mathsf{DTO} \to \mathsf{OCMon}$, which is left adjoint to the forgetful functor from OCMon to DTO. This exhibits $\mathsf{SListD}(A, \leqslant)$ as the free commutative monoid over A. The proofs are similar to the ones in Section 4, so we simply state the main result:

Proposition 32 (⚙). *Let A be a type with decidable equality and $\leqslant : A \to A \to \mathsf{Prop}$ a propositional decidable total order. Assuming function extensionality, $(\mathit{SListD}(A, \leqslant), \leqslant_L, \cup, \mathit{nil})$ is the free commutative monoid over A, i.e., $\mathit{SListD} : \mathsf{DTO} \to \mathsf{OCMon}$ forms a functor which is left adjoint to the forgetful functor $\mathcal{U} : \mathsf{OCMon} \to \mathsf{DTO}$ defined by $\mathcal{U}(X, \leqslant, \cdot, \epsilon) := (X, \leqslant)$.* □

Again we can get rid of the order relations if and only if we accept a little non-constructivity: The Ordering Principle holds if and only if both forgetful functors $\mathcal{U}_{\mathsf{DTO}} : \mathsf{DTO} \to \mathsf{Set}$ and $\mathcal{U}_{\mathsf{OCMon}} : \mathsf{OCMon} \to \mathsf{CMon}$ are equivalences.

6 Notions of Freshness for Other Free Structures

There are other notions of freshness relations that one can consider. These give rise to many other familiar free structures, some of which we consider here.

6.1 Free monoids

It is well known that free monoids can be represented as ordinary lists, with list concatenation as multiplication, and the empty list as the unit. A moment's thought gives that lists are the same thing as fresh lists with the constantly true relation as the freshness relation, i.e., when everything is fresh. Further, the category of sets equipped with their constantly true relation is isomorphic to the category of sets. We thus achieve the following theorem:

Proposition 33 (⚙). *Let $R_\top$ denote the complete relation on A. Then $\mathsf{List}\ A$ is isomorphic to $\mathit{FList}(A, R_\top)$, and hence, assuming function extensionality, $\mathit{FList}(A, R_\top)$ is the free monoid over the set A, i.e., $\mathit{FList}(-, R_\top) : \mathsf{Set} \to \mathsf{Mon}$ forms a functor which is left adjoint to the forgetful functor $\mathcal{U} : \mathsf{Mon} \to \mathsf{Set}$ defined by $\mathcal{U}(X, \cdot, \epsilon) := X$.* □

6.2 Free pointed sets

If we instead choose the constantly false relation, then we can only construct lists of lengths at most 1: creating a two-element list would require a proof that the first element is "fresh" for the second, i.e., a proof of falsity. This means that fresh lists for this relation gives rise to free pointed sets: elements can be included as singleton lists, and there is a new canonical point, namely the empty list. This is nothing but the *Maybe monad* in disguise! The category of sets equipped with their constantly false relation is again isomorphic to the category of sets, and writing $\mathsf{Set}_\bullet$ for the category of pointed sets, we have:

Proposition 34 (⚙). *Let $R_\bot$ denote the empty relation on A. Then $\mathsf{Maybe}\ A$ is isomorphic to $\mathit{FList}(A, R_\bot)$, and hence, assuming function extensionality, $\mathit{FList}(A, R_\bot)$ is the free pointed set over the set A, i.e., $\mathit{FList}(-, R_\bot) : \mathsf{Set} \to \mathsf{Set}_\bullet$ forms a functor which is left adjoint to the forgetful functor $\mathcal{U} : \mathsf{Set}_\bullet \to \mathsf{Set}$ defined by $\mathcal{U}(X, x) := X$.* □

6.3 Free Left-Regular Band Monoids

What kind of free structure do we get if we consider C. Coquand's original use of fresh lists for the inequality relation, or, more generally, for an apartness relation? Recall that an apartness relation $\not\approx\colon A \to A \to \mathsf{Prop}$ is a binary propositional relation satisfying axioms dual to those of an equivalence relation:

- *irreflexivity*: for all $x : A$, we do not have $x \not\approx x$;
- *symmetry*: for all $x, y : A$, if $y \not\approx x$ then $x \not\approx y$; and
- *cotransitivity*: for all $x, y, z : A$, if $x \not\approx y$, then $x \not\approx z$ or $z \not\approx y$.

An apartness relation $\not\approx$ is *tight*, if $\neg x \not\approx y \to x = y$. For any type X, there is a canonical "denial inequality" apartness relation $\neq\colon X \to X \to \mathsf{Type}$ given by $x \neq y := \neg(x = y)$ (which is tight if X has decidable equality), but there are often other more informative apartness relations for specific types.

In a fresh list where the notion of freshness is given by an apartness relation, it is thus indeed the case that if $x \#_{\not\approx} xs$, then x does not occur in xs due to the irreflexivity axiom. One might think that this should give rise to idempotent monoids, but in fact an even stronger axiom is satisfied, which allows to cancel a second occurrence of the same element also when there is an arbitrary number of elements between the occurrences. Such monoids are known as *left regular band monoids* [7] (and also as graphic monoids [20]).

Definition 35 (⚙). *A left-regular band monoid is a monoid $(X, \cdot, \epsilon)$, such that for any $x, y : X$, we have $x \cdot y \cdot x = x \cdot y$.*

Of course, a left-regular band monoid is in particular idempotent, since

$$x \cdot x = x \cdot \epsilon \cdot x \overset{\mathrm{LR}}{=} x \cdot \epsilon = x$$

for any $x : X$. We will now show that fresh lists for a decidable tight apartness relation gives rises to left-regular band monoids, again equipped with a decidable tight apartness relation. An apartness relation $\not\approx$ is tight and decidable if and only if for any $x, y : A$, we have either $x = y$ or $x \not\approx y$ — we will need this property to be able to remove elements from lists. Types equipped with a decidable tight apartness relation form a category $\mathsf{Type}_{\text{dec-apart}}$, whose morphisms are functions between the underlying types. Note that due to the decidability of the apartness relation, the underlying type also has decidable equality, and hence is in fact a set. Similarly, left-regular monoids equipped with apartness relations form a category $\mathsf{LRMon}_{\text{apart}}$, whose morphisms are monoid homomorphisms.

Proposition 36 (⚙). *Let A be a type and $\not\approx\colon A \to A \to \mathsf{Prop}$ a decidable tight apartness relation. Assuming function extensionality, $(\mathit{FList}(A, \not\approx), \neq)$ is the free left regular band monoid with a decidable tight apartness relation over the apartness type $(A, \not\approx)$, i.e., $\mathit{FList} : \mathsf{Type}_{\mathit{dec\text{-}apart}} \to \mathsf{LRMon}_{\mathit{apart}}$ forms a functor which is left adjoint to the forgetful functor $\mathcal{U} : \mathsf{LRMon}_{\mathit{apart}} \to \mathsf{Type}_{\mathit{dec\text{-}apart}}$ defined by $\mathcal{U}(X, \cdot, \epsilon, \not\approx) := (X, \not\approx)$.*

Proof. To construct a monoid operation on $\mathsf{FList}(A, \not\approx)$, we first use tightness and decidability of $\not\approx$ to define element removal $-\backslash\{-\} : \mathsf{FList}(A, \not\approx) \to A \to \mathsf{FList}(A, \not\approx)$, with $\mathsf{nil}\backslash\{x\} = \mathsf{nil}$, and

$$(\mathsf{cons}\, y\, ys\, p)\backslash\{x\} = \begin{cases} ys & \text{if } x = y \\ \mathsf{cons}\, y\, (ys\backslash\{x\})\, (\backslash\text{-fresh}(p)) & \text{if } x \not\approx y \end{cases}$$

where $\backslash\text{-fresh} : y \#_{\not\approx} ys \to y \#_{\not\approx} (ys\backslash\{x\})$ is defined simultaneously. For each $zs : \mathsf{FList}(A, \not\approx)$ and $y : A$, we then prove $\backslash\text{-removes}(zs, y) : y \#_{\not\approx} (zs\backslash\{y\})$ by induction on zs. We define the monoid multiplication on $\mathsf{FList}(A, \not\approx)$ as follows:

$$\begin{aligned} \mathsf{nil} \,\cup\, ys &:= ys \\ (\mathsf{cons}\ x\ xs\ p) \,\cup\, ys &:= \mathsf{cons}\ x\ ((xs \,\cup\, ys)\backslash\{x\})\ (\backslash\text{-removes}(xs \,\cup\, ys, x)) \end{aligned}$$

Associativity and the left regular band identity $xs \cup ys \cup zs = xs \cup ys$ are proven by induction on the lists involved. Finally the adjunction is proven similarly to the other fresh lists adjunctions. □

6.4 Free Reflexive Partial Monoids

Next we consider fresh lists for the equality relation on a set A. After forming a singleton list, we can only extend it by adding more copies of the already existing element in the list. Such a fresh list is thus either the empty list, or consists of $n > 0$ copies of some element in A:

Lemma 37 (⚙). *Let A be a set. Fresh lists for the equality relation $\mathsf{FList}(A, =)$ are isomorphic to structures of the form $1 + (A \times \mathbb{N}^{>0})$.* □

All our previous instantiations have been, at the very least, monoids. But what is the correct notion of multiplication for $\mathsf{FList}(A, =)$? In particular, how should we define it for lists which contain different elements, for example $[a, a] \cdot [b]$? There is no sensible way to combine these lists to produce a fresh list — we would like the monoid multiplication to be undefined in such cases. This leads us to consider the notion of partial monoids [26] (also called pre-monoids [5]): monoid-like structures that come with a "definedness" predicate which tells us when two elements may be multiplied.

Definition 38 (⚙). *A partial monoid is a set $X : \mathsf{Set}$ together with a propositional relation $\_ \cdot \_ \downarrow : X \to X \to \mathsf{Prop}$, a dependent function $\mathsf{op} : (x, y : X) \to (x \cdot y \downarrow) \to X$, and an element $\epsilon : X$, such that the following axioms hold, where we write $x \cdot_p y$ for $\mathsf{op}\ x\ y\ p$.*

- identity*: For all $x : X$, we have $\iota_{x,\epsilon} : (x \cdot \epsilon \downarrow)$ and $\iota_{\epsilon,x} : (\epsilon \cdot x \downarrow)$, and $x \cdot_{\iota_{x,\epsilon}} \epsilon = x = \epsilon \cdot_{\iota_{\epsilon,x}} x$;*
- associativity*: For all $x, y, z : X$,*

$$(\Sigma(p : (y \cdot z \downarrow)).(x \cdot (y \cdot_p z) \downarrow)) \longleftrightarrow (\Sigma(q : (x \cdot y \downarrow)).((x \cdot_q y) \cdot z \downarrow))$$

and for all $p : (y \cdot z \downarrow)$, $p' : (x \cdot (y \cdot_p z) \downarrow)$, $q : (x \cdot y \downarrow)$, $q' : ((x \cdot_q y) \cdot z \downarrow)$, we have $x \cdot_{p'} (y \cdot_p z) = (x \cdot_q y) \cdot_{q'} z$.

A partial monoid is reflexive *if* $(x \cdot x \downarrow)$ *for all* $x : X$.

Using Lemma 37, it is now not hard to show that $\mathsf{FList}(A, =)$ is a reflexive partial monoid with $\mathsf{inl} *$ as unit, and $((\mathsf{inr}\,(x, n)) \cdot (\mathsf{inr}\,(y, m)) \downarrow)$ holding exactly when $x = y$, with $\mathsf{inr}\,(x, n) \cdot_{\mathsf{refl}} \mathsf{inr}\,(x, m) = \mathsf{inr}\,(x, n + m)$. To show that $\mathsf{FList}(A, =)$ is the *free* reflexive partial monoid, we need to be able to construct powers x^n in arbitrary reflexive partial monoids. For example, $x^3 = x \cdot (x \cdot x)$ is defined because $(x \cdot x) \cdot (x \cdot x)$ is defined by reflexivity, hence by associativity also $x \cdot (x \cdot (x \cdot x))$ is defined, and in particular $x^3 = x \cdot (x \cdot x)$ is defined. In the general case, we define x^n by induction on $n : \mathbb{N}$, and simultaneously prove that both $(x \cdot x^k \downarrow)$ and $(x^m \cdot x \downarrow)$ for all $k, m : \mathbb{N}$, as well as that $x \cdot x^\ell = x^\ell \cdot x$ for all $\ell : \mathbb{N}$.

A morphism between partial monoids is a function between the carriers that preserves definedness and operations. Reflexive partial monoids and their morphisms form a category RPMon, and we again obtain a free-forgetful adjunction:

Proposition 39 (⚙). *Let A be a set. Assuming function extensionality, the set $\mathsf{FList}(A, =)$ with definedness relation and operations as described above is the free reflexive partial monoid over A, i.e., $\mathsf{FList} : \mathsf{Set} \to \mathsf{RPMon}$ forms a functor which is left adjoint to the forgetful functor $\mathcal{U} : \mathsf{RPMon} \to \mathsf{Set}$.* □

7 Conclusions and Future Work

We have shown how finite sets and multisets can be realised as fresh lists in plain dependent type theory, resulting in a well-behaved theory with good computational properties such as decidable equality, and without resorting to setoids or higher inductive types. Our only requirement is that the type we start with can be equipped with an order relation — a strict total order for finite sets, and a non-strict one for finite multisets. However, as suggested by a reviewer, relative adjunctions [29] can perhaps be used to formulate a universal property also over unordered structures. We have also shown how many other free structures can be understood in this unifying framework, such as free monoids, free pointed sets, and free left-regular band monoids. Measuring the efficiency of for example deciding equality in our free structures is left as future work.

There are many more algebraic structures that could be studied from the point of view of fresh lists, such as Abelian groups. Free algebraic structures without associativity tend to correspond to variations on binary trees [8]; as such, it would make sense to also investigate notions of "fresh trees", or perhaps a general notion of freshness for containers [1]. It would also be interesting to pin down exactly in which sense SList realises a predicative finite power set functor in type theory. One future use for this could be a constructive framework for modal logics supporting verification algorithms that are correct by construction.

Acknowledgements We thank Guillaume Allais for interesting suggestions, Ezra Schoen for the idea to consider reflexive partial monoids in Section 6.4, and the referees for insightful comments and improvements. This work was supported by the Engineering and Physical Sciences Research Council [EP/W52394X/1]; the National Physical Laboratory; and the Leverhulme Trust [RPG-2020-232].

References

1. Abbott, M., Altenkirch, T., Ghani, N.: Categories of containers. In: Gordon, A.D. (ed.) Foundations of Software Science and Computational Structures (FoSSACS '03). Lecture Notes in Computer Science, vol. 2620, pp. 23–38. Springer (2003). https://doi.org/10.1007/3-540-36576-1_2
2. The Agda Community: Agda standard library (2023), https://github.com/agda/agda-stdlib
3. Appel, A.W., Leroy, X.: Efficient extensional binary tries. Journal of Automated Reasoning **67**(1), 8 (2023). https://doi.org/10.1007/s10817-022-09655-x
4. Barthe, G., Capretta, V., Pons, O.: Setoids in type theory. Journal of Functional Programming **13**(2), 261–293 (2003). https://doi.org/10.1017/S0956796802004501
5. Bessis, D.: The dual braid monoid. Annales scientifiques de l'Ecole normale supérieure **36**(5), 647–683 (2003). https://doi.org/10.1016/j.ansens.2003.01.001
6. Blizard, W.D.: The development of multiset theory. Modern Logic **1**(4), 319 – 352 (1991)
7. Brown, K.S.: Semigroups, rings, and Markov chains. Journal of Theoretical Probability **13**(3), 871–938 (2000). https://doi.org/10.1023/a:1007822931408
8. Bunkenburg, A.: The Boom hierarchy. In: O'Donnell, J.T., Hammond, K. (eds.) Proceedings of the 1993 Glasgow Workshop on Functional Programming. pp. 1–8. Springer (1994). https://doi.org/10.1007/978-1-4471-3236-3_1
9. Choudhury, V., Fiore, M.: Free commutative monoids in Homotopy Type Theory. In: Hsu, J., Tasson, C. (eds.) Mathematical Foundations of Programming Semantics (MFPS '22). Electronic Notes in Theoretical Informatics and Computer Science, vol. 1 (2023). https://doi.org/10.46298/entics.10492
10. Choudhury, V., Karwowski, J., Sabry, A.: Symmetries in reversible programming: From symmetric rig groupoids to reversible programming languages. Proceedings of the ACM on Programming Languages **6**(POPL), 1–32 (2022). https://doi.org/10.1145/3498667
11. Coquand, C.: A formalised proof of the soundness and completeness of a simply typed lambda-calculus with explicit substitutions. Higher Order Symbolic Computation **15**(1), 57–90 (2002). https://doi.org/10.1023/A:1019964114625
12. Dybjer, P.: A general formulation of simultaneous inductive-recursive definitions in type theory. Journal of Symbolic Logic **65**(2), 525–549 (2000). https://doi.org/10.2307/2586554
13. Frumin, D., Geuvers, H., Gondelman, L., Weide, N.v.d.: Finite sets in homotopy type theory. In: International Conference on Certified Programs and Proofs (CPP '18). pp. 201–214. Association for Computing Machinery (2018). https://doi.org/10.1145/3167085
14. Gylterud, H.R.: Multisets in type theory. Mathematical Proceedings of the Cambridge Philosophical Society **169**(1), 1–18 (2020). https://doi.org/10.1017/S0305004119000045
15. Hedberg, M.: A coherence theorem for Martin-Löf's type theory. Journal of Functional Programming **8**(4), 413–436 (1998). https://doi.org/10.1017/s0956796898003153
16. Hinze, R.: Generalizing generalized tries. Journal of Functional Programming **10**(4), 327–351 (2000). https://doi.org/10.1017/S0956796800003713
17. Hutton, G.: A tutorial on the universality and expressiveness of fold. Journal of Functional Programming **9**(4), 355–372 (1999). https://doi.org/10.1017/s0956796899003500

18. Jech, T.: The Axiom of Choice. North-Holland (1973)
19. Joram, P., Veltri, N.: Constructive final semantics of finite bags. In: Naumowicz, A., Thiemann, R. (eds.) Interactive Theorem Proving (ITP '23). Leibniz International Proceedings in Informatics (LIPIcs), vol. 268, pp. 20:1–20:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2023). https://doi.org/10.4230/LIPIcs.ITP.2023.20
20. Lawvere, F.W.: Display of graphics and their applications, as exemplified by 2-categories and the Hegelian "taco". In: Proceedings of the first international conference on algebraic methodology and software technology. pp. 51–74 (1989)
21. McBride, C.: How to keep your neighbours in order. In: Jeuring, J., Chakravarty, M.M.T. (eds.) International conference on Functional programming (ICFP '14). pp. 297–309. Association for Computing Machinery (2014). https://doi.org/10.1145/2628136.2628163
22. Nordvall Forsberg, F.: Inductive-inductive definitions. Ph.D. thesis, Swansea University (2013)
23. Norell, U.: Towards a practical programming language based on dependent type theory. Ph.D. thesis, Chalmers University of Technology (2007)
24. Piceghello, S.: Coherence for Monoidal and Symmetric Monoidal Groupoids in Homotopy Type Theory. Ph.D. thesis, The University of Bergen (2021)
25. Pincus, D.: The dense linear ordering principle. Journal of Symbolic Logic **62**(2), 438–456 (1997). https://doi.org/10.2307/2275540
26. Poinsot, L., Duchamp, G., Tollu, C.: Partial monoids: associativity and confluence. Journal of Pure and Applied Mathematics: Advances and Applications **3**(2), 265 – 285 (2010)
27. Streicher, T.: Investigations into intensional type theory. Habilitation thesis (1993)
28. Swan, A.: If every set has some irreflexive, extensional order, then excluded middle follows. Agda formalisation by Tom De Jong available at https://www.cs.bham.ac.uk/~mhe/TypeTopology/Ordinals.WellOrderingTaboo.html
29. Ulmer, F.: Properties of dense and relative adjoint functors. Journal of Algebra **8**(1), 77–95 (1968). https://doi.org/10.1016/0021-8693(68)90036-7
30. The Univalent Foundations Program: Homotopy Type Theory: Univalent Foundations of Mathematics. Institute for Advanced Study (2013), https://homotopytypetheory.org/book/
31. Watters, S., Nordvall Forsberg, F., Kupke, C.: Agda formalisation of "A Fresh Look at Commutativity: Free Algebraic Structures via Fresh Lists". https://doi.org/10.5281/zenodo.8357335 (2023)