Filter Equivariant Functions: A symmetric account of length-general extrapolation on lists

Owen Lewis* Goodfire AI
Neil Ghani* Kodamai
Andrew Dudzik Google DeepMind
Christos Perivolaropoulos Google DeepMind
Razvan Pascanu Google DeepMind
Petar Veličković Google DeepMind

Abstract

What should a function that *extrapolates* beyond known input/output examples look like? This is a tricky question to answer in general, as any function matching the outputs on those examples can in principle be a correct extrapolant. We argue that a "good" extrapolant should follow certain kinds of *rules*, and here we study a particularly appealing criterion for rule-following in list functions: that the function should behave predictably even when certain elements are *removed*. In functional programming, a standard way to express such removal operations is by using a filter function. Accordingly, our paper introduces a new semantic class of functions – the filter equivariant functions. We show that this class contains interesting examples, prove some basic theorems about it, and relate it to the well-known class of map equivariant functions. We also present a geometric account of filter equivariants, showing how they correspond naturally to certain simplicial structures. Our highlight result is the *amalgamation* algorithm, which constructs any filter-equivariant function's output by first studying how it behaves on sublists of the input, in a way that extrapolates perfectly.

1 Introduction

We want to mathematically characterise functions with certain kinds of *rule-following* behaviour. The nature of rule-following is a longstanding philosophical riddle: given some pattern of behavior, what exactly does it mean to extend or *extrapolate* it consistently? While intuitively clear, the notion of extrapolation has been shown to be difficult or impossible to define in general.

Given an example behaviour like reverse [2, 3] = [3, 2], why prefer the extrapolation reverse [2, 4, 3] = [3, 4, 2] over, say, reverse [2, 4, 3] = [2, 4, 3]? We identify cases where this question has a precise answer based on certain symmetries, without relying on any implied semantics of reverse.

A practical motivation for our work comes from machine learning. In 2025, it is difficult to find tasks that neural network models do not do well, but length-based generalisation remains one such case. For example, one can train a neural network model to operate on lists of length up to 20, but still expect to see failures of generalisation at length $20,000$, or even before. In many other domains – image, audio, etc – symmetries have been a crucial tool in supporting out-of-distribution generalisation. While we do not develop this line of work here, we propose that the symmetries we identify for list functions could play a similar role for length generalisation of neural networks.

*Work performed while the author was at Google DeepMind.

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: Differentiable Learning of Combinatorial Algorithms (DiffCoALG).

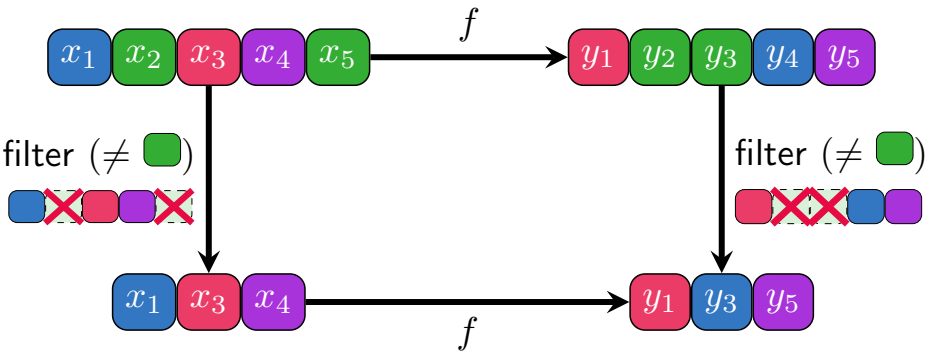

Figure 1: We leverage filter equivariance as a way to express functions which are length-invariant through removing items by value. A filter-equivariant (FE) function f may be composed with filter in any order, yielding the same results.

2 Symmetries of list functions, and filter equivariance

We study symmetries of list functions, i.e. functions of the form $f : [a] \to b$, mapping lists of elements of type a – which we denote using $[a]$ – to an output of type b.

As will be shown, the specific kind of symmetric behaviour we will study here – which we call *equivariance* – necessarily implies that $b = [a]$, i.e., the function must map from lists to lists, without changing the underlying type of the elements. This allows the composition of these functions in arbitrary order. We define list function equivariance as follows:

Definition 2.1 *We say that a list function $f : [a] \to [a]$ is **equivariant**[2] with respect to a particular transformation $g : [a] \to [a]$ if, when composing f with g, the composition order does not matter.*

Our work is not the first to study list function symmetries. In particular, a well studied symmetry is with respect to the map : $(a \to b) \to [a] \to [b]$ function. For a given element-wise function $\psi : a \to b$, map $\psi : [a] \to [b]$ applies ψ to each element of the input independently. It is a standard result that we can characterise all functions that commute with map as *natural transformations* from the list functor to itself, and the idea may be generalized to other functors too, such as trees. While map-equivariant functions are certainly important and elegant, they have no direct bearing on length-general extrapolation; as map does not change the length of the list it operates over. To go further in our objective, we need to study operators that *perturb a list's length*.

This can usually be done either by adding or removing elements from it. In general, *removals* tend to be easier to reason about, as they constrain the space of possible elements that can be transformed. Further, removals based on element *position* (e.g. removing the last element) would exclude several important rule-following functions from consideration, such as reverse and sort. This motivates us to consider *removals by value*.

2.1 Equivariance to filter

The canonical way to remove elements by value is to apply the function filter : $(a \to \text{Bool}) \to [a] \to [a]$ that selects just those elements from a list that match a predicate function $\phi : a \to \text{Bool}$:

$$\text{filter even } [1, 2, 3] = [2] \qquad \text{filter odd } [1, 2, 3, 4] = [1, 3]$$

For a function to be symmetric to all kinds of value-based removals, we need it to remain equivariant across all choices of predicates ϕ, as follows (see Figure 1):

[2]Readers familiar with geometric deep learning [1, GDL] will likely recognise that this definition of equivariance is *weaker* than usual, as g is not constrained by a *group* structure. This means it does not need to be a *lossless* transformation. In this sense, the functions we study here are more easily expressed by frameworks going beyond GDL, such as categorical deep learning [2, CDL].

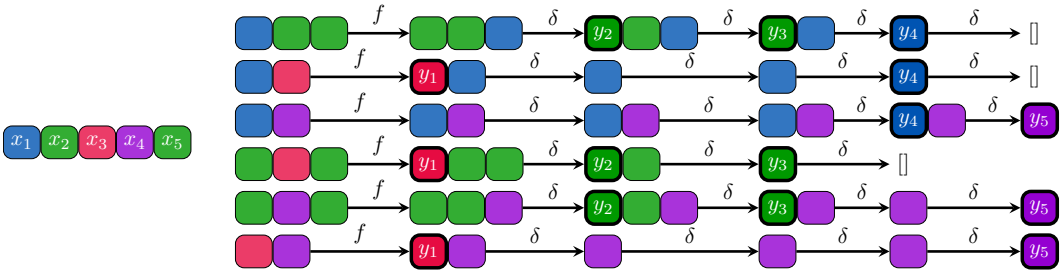

Figure 2: The process of amalgamating the output of $f\ [x_1, x_2, x_3, x_4, x_5] = [y_1, y_2, y_3, y_4, y_5]$, for the same filter-equivariant function f and input as presented in Figure 1.

Definition 2.2 *We say that a list function $f : [a] \to [a]$ is* filter-***equivariant*** *(FE) if, for all choices of $\phi : a \to$ Bool, composing f with* filter *ϕ gives the same result in either order:*

$$\forall \phi : a \to \mathsf{Bool}.\ (\mathsf{filter}\ \phi) \cdot f = f \cdot (\mathsf{filter}\ \phi) \tag{1}$$

Further, $f : [a] \to [a]$ is ***natural*** filter-***equivariant*** *(NFE) if it is both* map- *and* filter-*equivariant.*

It's worth briefly surveying a few prominent examples of (N)FEs. Namely, key NFEs are reverse and inflate n, where inflate : Nat $\to [a] \to [a]$ repeats each element in the list n times. These two functions turn out to be the only important basis functions for NFEs, as *all* other NFEs can be obtained by function composition and concatenation of these primitive functions. For reasons of brevity we cannot derive this here, but leave it to Appendix A[3].

Clearly, all NFEs are also FEs, but the converse does not hold. The sort function, as well as filter ϕ itself, are filter equivariant but not NFEs. Intuitively, map equivariants cannot depend on inputs' *values* – otherwise there would exist a ψ for which map ψ would modify their behaviour. An example of a function that is map equivariant, but is *not* FE is triangle, which repeats the ith element of an input list i times, e.g. triangle $[3, 7, 5] = [3, 7, 7, 5, 5, 5]$. This is due to the fact FEs cannot depend on elements' *absolute position* (which filter may easily displace).

3 Amalgamation

Now we demonstrate our key result, showing exactly how filter equivariance supports length generalisation. In particular, we show how an FE's behavior on any list is determined by its behaviour on all of that list's sublists with two unique elements.

For example, if a function $f : [a] \to [a]$ is FE, and we know the following:

$$f\ [2,1,2] = [1,2,2] \qquad f\ [3,2,2] = [2,2,3] \qquad f\ [3,1] = [1,3]$$

we can deduce $f\ [3,2,1,2] = [1,2,2,3]$—we *extrapolated* sort-like behaviour from small examples!

For a given input list xs, one needs to apply all necessary filters to reduce xs to two unique elements. Then, we compute f on these smaller lists, and 'glue' the results together. This process of gluing partial results together we call **amalgamation**, and it is depicted in Figure 2.

Firstly, we need a way to reason about these collections of filtered lists. Let X be the set of all possible values in our input list, and let $X \setminus zs$ be the set X with elements in $zs : [X]$ removed. Then we can reason about filtered collections as $\chi : [X] \to [X]$ such that $\chi\ zs \in [X \setminus zs]$. Intuitively, $\chi\ zs$ may represent the result of f when filtering out elements in zs, e.g. $\chi\ zs = f\ (\mathsf{filter}\ (\notin zs)\ xs)$.

Clearly, not every such collection of lists can be amalgamated into one unique list. Therefore, we need to carve out a subtype of $[X] \to [X \setminus zs]$ containing only *amalgamable* collections—collections for which only one, unique and correct way to amalgamate exists.

[3]We can also reason about the more general case of FEs, but the derivation is less clean—see Appendix B.

We can define this subtype recursively. Firstly, we can define the predicate $\mathsf{AM}_0\ X$, of collections, $\chi : [X] \to [X \backslash zs]$, from which the first element, $x_0 : X$, is amalgamable:

$$\chi \in \mathsf{AM}_0\ X \iff \exists x_0 \in X.\, \forall zs : [X].\, x_0 \notin zs \implies \mathsf{head}\ (\chi\ zs) = x_0,$$

where $\mathsf{head} : [a] \to a$ returns the first element ("head") of its input list. This means that all lists in the collection that still have x_0 must put it at the first position.

Now, we must ensure the amalgamability is also maintained across the rest of the list, and using AM_0, we can ground a recursive procedure to do so. We define a new predicate, $\mathsf{AM}\ X$, by:

$$\begin{aligned} \chi \in \mathsf{AM}\ X \quad \iff \quad & (\forall zs : [X].\, \chi\ zs = []) \\ & \vee\ (\chi \in \mathsf{AM}_0\ X \wedge\ \delta\chi \in \mathsf{AM}\ X) \end{aligned}$$

where $\delta\chi : [X] \to [X \backslash zs]$ is defined as follows:

$$\delta\chi\ zs = \begin{cases} \chi\ zs & x_0(\chi) \in zs \\ \mathsf{tail}\ (\chi\ zs) & x_0(\chi) \notin zs \end{cases}$$

where $x_0(\chi)$ is the unique head element of xs (which must exist if $\chi \in \mathsf{AM}_0\ X$). The definition of $\mathsf{AM}\ X$ entails that amalgamable collections *either* have no elements left to amalgamate, *or* they can uniquely amalgamate the first element x_0, and the remaining collection of lists, $\delta\chi$ – obtained by removing $x_0(\chi)$ from the head of all relevant lists, by the use of tail – is amalgamable.

Using this definition, we can establish an isomorphism between $[X]$ and $\mathsf{AM}\ X$, via the function $\mathsf{amal} : \mathsf{AM}\ X \to [X]$ which repeatedly extracts the first element until convergence:

$$\mathsf{amal}\ \chi = \begin{cases} [] & \forall zs : [X].\, \chi\ zs = [] \\ x_0(\chi) :: \mathsf{amal}\ \delta\chi & \text{otherwise} \end{cases} \tag{2}$$

This can be interpreted as iteratively searching for $x_0(\chi)$ by majority voting across the first elements of the outputs of f, as in the following pseudocode:

Input: filter-equivariant $f : [X] \to [X]$, $\quad xs : X$, $\quad |X| \geq 3$
$\chi \leftarrow []$
for $X' \subset X$ *s.t.* $|X'| = |X| - 2$ **do**
  $\chi \leftarrow (f\ (\mathsf{filter}\ (\notin X')\ xs)) :: \chi$
end
$ys \leftarrow []$
while $\exists l \in \chi$ *s.t.* $\mathsf{len}\ l > 0$ **do**
  for $x \in X$ **do**
    $\mathrm{score}(x) \leftarrow 0$
  end
  for $l \in \chi$ **do**
    if $\mathsf{len}\ l > 0$ **then**
      $\mathrm{score}(\mathsf{head}\ l) \leftarrow \mathrm{score}(\mathsf{head}\ l) + 1$
    end
  end
  $x_0 \leftarrow \arg\max_{x \in X} \mathsf{score}(x)$
  $ys \leftarrow x_0 :: ys$
  for $l \in \chi$ **do**
    if $\mathsf{len}\ l > 0$ **then**
      if $\mathsf{head}\ l = x_0$ **then**
        $l \leftarrow \mathsf{tail}\ l$
      end
    end
  end
end
return reverse ys

With this framework in place, all that remains is to prove that applying f to all sublists of xs with two unique elements forms an amalgamable collection—this is our key theorem, which we prove in full.

4 Outputs on lists of two unique elements are amalgamable

To lay the groundwork for proving our work's key Theorem, we first establish an isomorphism between $[X]$ and $\mathsf{AM}\ X$. Specifically, in one direction, we can map lists in X to amalgamable collections in $\mathsf{AM}\ X$ by way of the *filter collection*, $\pi : [X] \to X \to [X \setminus x]$, defined by $\pi\ xs\ x = \mathsf{filter}\ (\neq x)\ xs$. In the other direction, as already hinted in the main text, we can invert this function by amalgamating—so long as the input list $xs : [X]$ has at least three elements:

Lemma 4.1 *Let $xs : [X]$ be a list with at least three unique elements. Then $\pi\ xs \in \mathsf{AM}\ X$. Further, π is an isomorphism with* amal *as its inverse.*

Proof: The first statement (on amalgability of $\pi\ xs$) holds by induction on xs, noting that $\pi \cdot \mathsf{tail} = \delta \cdot \pi$. This statement can also be used to prove $\mathsf{amal} \cdot \pi = \mathsf{identity}$. □

It will also be useful to note that this same logic allows us to amalgamate $\pi\ (f\ xs)$, for any filter-equivariant $f : [X] \to [X]$:

Lemma 4.2 *Let $f : [X] \to [X]$ be a* filter*-equivariant function, and $xs : |X|$ be a list with at least three unique elements. Then, $\pi\ (f\ xs) \in \mathsf{AM}\ X$.*

Proof: If f does not reduce the size of xs – i.e., $|xs| = |f\ xs|$ – then surely $f\ xs$ has at least three unique elements, and Lemma 4.1 applies. If, instead, $|xs| > |f\ xs|$ – for example, if f is a filter – we need to apply more care if $|f\ xs| < 3$.

When $|f\ xs| = 0$, $f\ xs = []$, therefore $(\pi\ (f\ xs))\ x = []$ for all $x : X$, and hence $\pi\ (f\ xs) \in \mathsf{AM}\ X$.

Otherwise, let $\chi\ x = \pi\ (f\ xs)$ be our collection of outputs of f. We first show that $x_0(\chi) = \mathsf{head}\ (f\ xs)$. Now, since $\chi\ x = \mathsf{filter}\ (\neq x)\ (f\ xs)$, we can conclude that $\mathsf{head}\ (\chi\ x) = \mathsf{head}\ (f\ xs)$ whenever $x \neq \mathsf{head}\ (f\ xs)$. And since $|X|$ – the number of collections in χ – is at least 3, this element is uniquely specified. Any other candidate x' for $x_0(\chi)$ would not be the head of the list $\chi\ z$, for $z \neq x', z \neq x$ (which must exist since $|X| \geq 3$). Therefore, $\pi\ (f\ xs) \in \mathsf{AM}_0\ X$

It is possible to show (just like for Lemma 4.1) that $\pi \cdot \mathsf{tail} = \delta \cdot \pi$, and we are done. □

Now we can use this to show our key result:

Theorem 4.3 *Let $f : [X] \to [X]$ be* filter*-equivariant, and $xs : [X]$ be a list with at least three unique elements. Then,* amal *can perfectly reconstruct $f\ xs$ from knowledge of the collection $\{f\ ys\}$, where ys ranges over all sublists of xs with two unique elements.*

Proof: First, we show that we can correctly amalgamate $f\ xs$ from the collection of all outputs of f where one input element had been filtered out, i.e., $\{f\ (\mathsf{filter}\ (\neq x)\ xs)\}_{x:X}$:

$$f\ xs = \mathsf{amal}\ (\pi\ (f\ xs)) = \mathsf{amal}\ (\lambda x \to \mathsf{filter}\ (\neq x)\ (f\ xs)) = \mathsf{amal}\ (\lambda x \to f\ (\mathsf{filter}\ (\neq x)\ xs))$$

where we exploited Lemma 4.2 in the first step, and the fact that f is filter-equivariant in the end.

Now we observe that, if $\mathsf{filter}\ (\neq x)\ xs$ has more than two unique elements, those can be themselves reconstructed from collections obtained by filtering an additional element, using exactly the same argument:

$$f\ xs = \mathsf{amal}\ \left(\lambda x \to \begin{cases} f\ (\mathsf{filter}\ (\neq x)\ xs) & |\mathsf{filter}\ (\neq x)\ xs| \leq 2 \\ \mathsf{amal}\ (\lambda y \to f\ (\mathsf{filter}\ (\neq y)\ (\mathsf{filter}\ (\neq x)\ xs))) & \text{otherwise} \end{cases}\right)$$

This is sufficient to prove our Theorem – as we continue to stack nested filter and amal calls until we have only lists of no more than two unique elements, from which we can reconstruct everything needed. However, it's arguably not very satisfying, as it requires multiple nested calls to amal.

Conveniently, we can prove a lemma that implies it is sufficient to just amalgamate once:

Lemma 4.4 *Let $f : [X] \to [X]$ be* filter *equivariant, and $xs : [X]$ be a list with at least four unique elements. Then it is true that:*

$$\mathsf{amal}\ (\lambda x \to \mathsf{amal}\ (\lambda y \to f\ (\mathsf{filter}\ (\neq \{x, y\})\ xs))) = \mathsf{amal}\ (\lambda(x, y) \to f\ (\mathsf{filter}\ (\neq \{x, y\})\ xs))$$

where, in the right hand side, we set $X' = X \times X$ and $\chi' : X' \to [X]$ to represent our collection of lists, indexed over tuples (x, y) where $x \neq y$.

Proof: We have already showed that the left-hand side is equal to $f\ xs$. First, we can show that the first element of the right-hand side matches this, i.e., $x_0(\chi') = \mathsf{head}\ (f\ xs)$.

Whenever $\mathsf{head}\ (f\ xs) \neq x \wedge \mathsf{head}\ (f\ xs) \neq y$, $\mathsf{head}\ (f\ (\mathsf{filter}\ (\neq \{x, y\})\ xs)) = \mathsf{head}\ (f\ xs)$, which satisfies the constraints needed for $x_0(\chi')$. The only thing left is to show it is unique. For any other candidate head $x' \neq \mathsf{head}\ (f\ xs)$, it will not be the head element in all lists where it appears – particularly, any list indexed by (x', y') where $y' \neq \mathsf{head}\ (f\ xs)$. Therefore, it must hold that $x_0(\chi') = \mathsf{head}\ (f\ xs)$.

Now, we must show that applying $\delta\chi'$ allows us to continue amalgamating in a way that gradually reconstructs $f\ xs$. Note that, because χ' uses a slightly modified input space, we need to redefine how δ applies here, by simply extending the check of which lists contain $x_0(\chi')$:

$$\delta\chi'\ (x, y) = \begin{cases} \chi'\ (x, y) & x = x_0(\chi') \vee y = x_0(\chi') \\ \mathsf{tail}\ (\chi'\ (x, y)) & x \neq x_0(\chi') \wedge y \neq x_0(\chi') \end{cases}$$

Now, we can show that this collection is equivalent to the collections we would get by processing the tail of our desired list with the same two-element filtering, i.e.:

$$\delta\chi'\ (x, y) = \mathsf{filter}\ (\neq \{x, y\})\ (\mathsf{tail}\ (f\ xs))$$

To see this, let us examine both cases: when $x = x_0(\chi') \vee y = x_0(\chi')$, then we return $\chi'\ (x, y) = \mathsf{filter}\ (\neq \{x, y\})\ (f\ xs) = \mathsf{filter}\ (\neq \{x, y\})\ \mathsf{tail}\ (f\ xs)$, since the head element of $(f\ xs)$ will be filtered out in this case. When $x \neq x_0(\chi') \wedge y \neq x_0(\chi')$, then we return $\mathsf{tail}\ (\chi'\ (x, y)) = \mathsf{tail}\ (\mathsf{filter}\ (\neq \{x, y\})\ (f\ xs)) = \mathsf{filter}\ (\neq \{x, y\})\ (\mathsf{tail}\ (f\ xs))$, where the final swap is possible because we know the first element of $f\ xs$ will not be filtered out by the filter call.

Now, it holds (by induction on $f\ xs$, as in Lemma 4.1) that $\mathsf{amal}\ \delta\chi' = \mathsf{tail}\ (f\ xs)$, and we are done.

4.1 NFEs are amalgamable from a *single* example

Our arguments above show that any FE function output $f\ xs$ is determined by $\{f\ ys\}$, where ys ranges over all sublists of xs with two unique elements. To specialise this to NFEs, first recall from Appendix A.1 that we can assume an input list to an NFE has unique elements – if it doesn't already, we can uniquify with enumerate without changing f's behavior. Thus, this section's "sublists with two unique elements" become *precisely* the sublists of length two.

Further, given a length-two list $[x, y]$ with $x \neq y$, for any other length-two list $[p, q]$, there exists some function $g : X \to X$ with $[p, q] = \mathsf{map}\ g\ [x, y]$. Then, map equivariance implies that $f\ [p, q] = (\mathsf{map}\ g)\ (f\ [x, y])$, meaning that if we know a *single* length-two example $f\ [x, y]$ it suffices to determine *every other* length-two example, and hence to determine f's global behavior by amalgamation. Hence, if we know our function is an NFE, we can reconstruct its behaviour on *any* input from observing just a single example!

A brief counterexample shows how analogous reasoning fails for general FEs. Consider a function $f : [a] \to [a]$ that, for each unique element $x \in xs$ postpends $|x|^2$ copies of x to an initially-empty output list, where $|x|$ is the number of times x appears in xs. For example,

$$f\ \ [4, 7, 4, 7, 8] = [4, 4, 4, 4, 7, 7, 7, 7, 8]$$

We can see that f is not map equivariant because, for example, it does not commute with mapping a constant function. Further, we can see that f acts as the identity function on *any* two-element list with unique elements, and hence its general behavior cannot be deduced by such examples.

References

[1] Michael M Bronstein, Joan Bruna, Taco Cohen, and Petar Veličković. Geometric deep learning: Grids, groups, graphs, geodesics, and gauges. *arXiv preprint arXiv:2104.13478*, 2021.

[2] Bruno Gavranović, Paul Lessard, Andrew Joseph Dudzik, Tamara Von Glehn, João Guilherme Madeira Araújo, and Petar Veličković. Position: Categorical deep learning is an algebraic theory of all architectures. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp, editors, *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 15209–15241. PMLR, 21–27 Jul 2024.

A Characterising all NFEs

As hinted in the main body of the paper, there is a neat way in which we can express the collection of *all* natural filter equivariant functions (NFEs) – this Appendix explores this in greater depth.

Firstly, we can create new (N)FEs by combining existing ones via *concatenation* and *composition*. In other words, (N)FEs possess the following two monoid structures:

Lemma A.1 *NFEs (and FEs) form a monoid with unit being the constant function mapping every list to the empty list, and with addition defined by pointwise list concatenation:*

$$(f +\!\!+ g)\ xs = (f\ xs) +\!\!+ (g\ xs)$$

Lemma A.2 *NFEs (and FEs) form another monoid with the unit being the identity function and the composition of two (N)FEs giving an (N)FE.*

Given a list $xs :: [a]$, we denote by $|xs|$ the elements occurring in xs. This function may be defined, for example, in Haskell, as follows:

```
|-| :: [a] -> [a]
| [] | = []
|x:xs| = x : | filter (/= x) xs |
```

Note that $|-| : [a] \to [a]$ is FE but not NFE. Our first insight is that FE functions do not add any new values to the input list. (As a corollary, FE functions must map the empty list to the empty list.)

Lemma A.3 *Let $f : [a] \to [a]$ be* filter*-equivariant. Then, for all input lists $xs : [a]$, $|f\ xs| \subseteq |xs|$.*

Proof: Let xs be a list, assume $y \notin |xs|$ and we can show $y \notin |f\ xs|$. Define the predicate ϕ by $\phi\ z := (z \neq y)$. Then $f\ xs = (f \cdot \mathsf{filter}\ \phi)\ xs = (\mathsf{filter}\ \phi \cdot f)\ xs = \mathsf{filter}\ \phi\ (f\ xs)$. Therefore, $y \notin |f\ xs|$.

A.1 Properties of (k-)NFE functions

Recall that natural filter equivariants (NFEs) are functions that are equivariant to both map and filter. This is a very strong pair of constraints: it implies that the function must both behave predictably across various lengths *and* its behaviour must not depend on the values of the items inside the list. We begin by studying NFEs, as these constraints will more easily yield interesting mathematical structure.

As a general note for what follows, we may assume without loss of generality that the input list to any NFE has unique elements. One way to force the elements of an input list xs to be unique by "tagging" each element with its index:

$$\mathsf{enumerate}\ [x_0, x_1, \ldots, x_n] = [(x_0, 0), (x_1, 1), \ldots, (x_n, n)]$$

This operation can be undone by:

$$\mathsf{unenumerate}\ [(x_0, 0), (x_1, 1), \ldots, (x_n, n)] = [x_0, x_1, \ldots, x_n]$$

Noting that $\mathsf{unenumerate} = \mathsf{map\ first}$, we can see that uniquification of input elements preserves the operation of any natural transformation f, in the following sense:

$$\begin{aligned}\mathsf{unenumerate} \cdot f \cdot \mathsf{enumerate} &= (\mathsf{map\ first}) \cdot f \cdot \mathsf{enumerate} \\ &= f \cdot (\mathsf{map\ first}) \cdot \mathsf{enumerate} \\ &= f \cdot \mathsf{unenumerate} \cdot \mathsf{enumerate} \\ &= f\end{aligned}$$

The first result we show for NFEs is that they must replicate each element a consistent number of times—if a particular element is replicated k times by an NFE, *all* of them must be replicated k times.

Lemma A.4 *Let $f : [a] \to [a]$ be natural* filter*-equivariant and let $k = \mathsf{len}(f\ [*])$ where $* : 1$. Then, for all inputs $xs : [a]$, we have* $\mathsf{len}\ (f\ xs) = k \times (\mathsf{len}\ xs)$*. Further, if x occurs m times in xs, then x occurs $k \times m$ times in $f\ xs$.*

Proof: First, we use the map equivariance of f to show that, when applied to any singleton list, it must return a list of length k. Further, we know – due to Lemma A.3 – that this list can only contain the singleton element in the original input list. Now, we can use the filter equivariance of f to extend the result for singletons to arbitrary lists with no repeated elements. For the case where the input list contains repeated elements, we can apply enumerate as described above, apply the previous result, and then use map equivariance to conclude the proof.

We call a function $f : [a] \to [a]$ a k-NFE if it is an NFE that maps singletons to lists of length k. We first describe 1-NFEs and then tackle the general case. Since Lemma A.4 implies that 1-NFEs do not modify the overall collection of values in the input, they are nothing more than a pre-defined *permutation* of the input elements—a standard example of which is reverse. By map equivariance, a 1-NFE must therefore be given by a family of permutations $t : \prod_{n:\mathsf{Nat}} \mathsf{Perm}(n)$ where Perm maps every natural number n to the set of permutations on the set $\{0, .., n-1\}$. For the specific case of reverse, these permutations are

$$t_{\mathsf{reverse}} = [[], [0], [1, 0], [2, 1, 0], [3, 2, 1, 0] \ldots]$$

While this approach will allow us to define 1-NFEs, we still need to include the coherence conditions relating the various permutations ($t\ 0$ to $t\ 1$ to $t\ 2$, etc.). This is the essence of length-generality: the permutations at longer lengths need to be systematically related to the ones at shorter lengths. To define this coherence, we leverage simplicial algebra:

Definition A.5 *Let Δ be the semi-simplicial category. Its objects are the finite natural numbers (thought of as sets $\{0, 1, \ldots, n-1\}$ for $n : \mathsf{Nat}$), and its morphisms $n \to m$ are inclusions. A* ***semi-simplical set*** *is a functor $F : \Delta^{\mathrm{op}} \to \mathsf{Set}$, i.e. a family of sets $F(n)$ with functions $F(f) : F(m) \to F(n)$ for every inclusion $f : n \to m$, obeying the functoriality laws.*

With the concept of semi-simplicial sets handy, we can define two particularly important semi-simplicial sets: the *terminal* set and the *permutation* set:

Example A.6 *The terminal semi-simplicial set, 1, is the functor that maps every object to the one-element set. The functor* Perm *is the semi-simplicial set that maps n to the set $\mathsf{Perm}(n)$ of permutations on n. Given any inclusion $f : n \to m$, define $\mathsf{Perm}(f) : \mathsf{Perm}(m) \to \mathsf{Perm}(n)$ to map a permutation on m elements to the permutation on n elements, by simply removing the elements in m that are not in n.*

Using these two semi-simplicial sets, we can formally, coherently define a 1-NFE:

Lemma A.7 *A 1-NFE is exactly the natural transformation* $1 \to \mathsf{Perm}$*. That is, a 1-NFE is a family of permutations* $t_n : \mathsf{Perm}(n)$ *such that for every inclusion* $f : n \to m$*,* $t_n = \mathsf{Perm}(f)\ t_m$*. We call such families* ***semi-simplicial permutations****.*

Indeed, those familiar with the terminology of category theory will note that a 1-NFE is simply a *cone* over the functor Perm and that the set of 1-NFEs is $\lim_{n\to\infty} \mathsf{Perm}(n)$. Below we will refer to families like this as cones, for brevity.

Next, we turn to the general case of k-NFEs, where a similar story can be told. We know – by Lemma A.4 – that a k-NFE maps lists of length n to lists of length $k \times n$, by mapping a list xs to a permutation of $\mathsf{inflate}\ k\ xs$. Because of the map equivariance of NFEs, this data is given through a permutation of $k \times n$, i.e., an element of $\mathsf{Perm}(k \times n)$. Thus, we define:

Definition A.8 *The semi-simplicial set of* k*-permutations is defined by the functor*

$$\Delta^{\mathrm{op}} \xrightarrow{-\times k} \Delta^{\mathrm{op}} \xrightarrow{\mathsf{Perm}} \mathsf{Set}$$

A cone for the above functor is called a k***-semi-simplicial permutation****.*

Using k-semi-simplicial permutations, we can extend our definition of 1-NFEs to k-NFEs analogously:

Lemma A.9 *A particular* k*-NFE is a* k*-semi-simplicial permutation, and hence the set of all* k*-NFEs is* $\lim_{n\to\infty} \mathsf{Perm}(n \times k)$*.*

Just as natural transformations give a categorical interpretation of map equivariance, semi-semplicial permutations develop the category theory for filter equivariance. In the next section, we turn to type theory.

A.2 Inductive characterisation of NFEs

Given the relationship between k-semi-simplicial permutations and 1-semi-simplicial permutations, one may wonder if k-NFEs can be built from 1-NFEs. A positive answer would hint at a compositional semantics for NFEs where k-NFEs are built from k'-NFEs for $k' \leq k$.

This claim turns out to be true, and it is possible to provide an *inductive* presentation of the set of NFEs as an *inductive data type*. Concretely, in Haskell, we have:

```
data NFE = Z
         | P Nat NFE
         | N Nat NFE
```

To be clear, the data type `NFE` contains only the unique representations of NFEs [4] and is a drastic simplification of the *formal* definition of NFEs, which consists of data (as above) which must obey equivariance constraints (missing from the above).

Rather than providing NFEs as an inductive data type, we could give k-NFEs a definition as an inductive family, by defining $\mathsf{NFE} : \mathsf{Nat} \to \mathsf{Set}$ mapping k to the set of k-NFEs. The only reason why we chose not to do so is because we have currently found no use for such a presentation.

To understand the inductive type `NFE`, consider it as a defining lists of natural numbers with two different ways to construct them. Indeed,

Lemma A.10 *There is a bijection between* NFE *and* $\mathsf{List}(\mathsf{Nat} + \mathsf{Nat})$*.*

[4] The constructors `P` and `N` are assumed to have their first input as $n : \mathsf{Nat}$, where $n \geq 1$.

We now explain how `NFE` represents an NFE by giving a function mapping every element of the data type `NFE` to an actual NFE. Since `NFE` is the free monad on $\mathsf{Nat} + \mathsf{Nat}$, and NFEs form a monoid, it suffices to map $\mathsf{Nat} + \mathsf{Nat}$ to the class of NFEs. We do this by sending an element n of the first injection to the NFE $\mathsf{inflate}\ n$ and an element n of the second injection to $\mathsf{reverse} \cdot (\mathsf{inflate}\ n)$. This gives the following function

```
[[ - ]] :: NFE -> [a] -> [a]
[[ Z ]] xs     = []
[[ P n m ]] xs = inflate n xs ++ [[ m ]] xs
[[ N n m ]] xs = reverse (inflate n xs) ++ [[ m ]] xs
```

That every element of `NFE` generates an NFE is clear, since we have already seen that NFEs contain the constant function returning the empty list `[]`, contain $\mathsf{reverse}$ and $\mathsf{inflate}\ n$, and are closed under function composition and under concatenation. To see these are the *only* NFEs, we can first leverage map equivariance to show that the only 1-NFEs are the $\mathsf{identity}$ and $\mathsf{reverse}$.

This result scales to k-NFEs by enumeration across k; it further shows the following:

Lemma A.11 *The number of k-NFEs is $2 \times 3^{k-1}$ when $k \geq 1$.*

For example, the only 2-NFEs are $\mathsf{inflate}\ 2$, $\mathsf{reverse} \cdot (\mathsf{inflate}\ 2)$, $\mathsf{identity} + \mathsf{identity}$, $\mathsf{reverse} + \mathsf{identity}$, $\mathsf{identity} + \mathsf{reverse}$, and $\mathsf{reverse} + \mathsf{reverse}$ – six functions in total.

B Characterising filter equivariants

Having established the foundations that allowed us to characterise *all* NFEs, we are ready to relax the map equivariance constraint, and refine our analysis for the case of FEs – the crux of our paper.

Recall how we characterised NFEs by the fact they will *inflate* the input list by a certain constant k (leading to k-NFEs). It is possible to generalise this to the FE case—however, when doing so, not all elements will necessarily be replicated by the same amount, leading to the following result:

Lemma B.1 *Given a filter-equivariant function $f : [a] \to [a]$, there is an underlying function $\Phi : a \to \mathsf{Nat} \to \mathsf{Nat}$ such that if x occurs n times in xs, then x occurs $\Phi(x, n)$ times in $f\ xs$*

Proof: The proof of the above defines

$$\Phi(x, n) = \mathsf{len}(f\ (\mathsf{repeat}\ n\ x))$$

To prove the lemma, use filter equivariance for the specific function $\mathsf{filter}\ \phi$ keeping only the element of interest, i.e., $\phi\ z := (z = x)$. □

With knowledge of the occurrence function Φ, we can characterise the final output of any FE as follows:

Lemma B.2 *Let f be filter-equivariant. Then, if the distinct elements of xs are $[x_1, \ldots, x_n]$ and x_i occurs n_i times in xs, then $f\ xs$ is a permutation of*

$$\mathsf{concat}\ [\ \mathsf{repeat}\ \Phi(x_i, n_i)\ x_i\ \mid\ 1 \leq i \leq n\ ]$$

These permutations must be coherent with respect to the same semi-simplicial structure used to describe NFEs. To formalise this statement, we can generalise the idea of k-NFEs to Φ-FEs, where $\Phi : a \to \mathsf{Nat} \to \mathsf{Nat}$ is the corresponding occurrence function of the FE.

We characterised k-NFEs via functors whose domain was the semi-simplicial category Δ, because all we needed to know was the length of the list – map equivariance meant we could assume, without loss of generality, that all elements were distinct. This is not the case for Φ-FEs where we need to know *which elements are in the input list* and *what is their multiplicity* (the two inputs for Φ). This is a setting suitable for *multisets* (also known as *bags*), and hence we define category Bag as follows

Definition B.3 *The category* Bag *is defined as follows*

- *Objects are tuples* (n, f) *with finite numbers* $n : \mathsf{Nat}$ *and functions* $f : \mathsf{Fin}\ n \to \mathsf{Nat}$.[5]
- *Morphisms* $(n, f) \to (n', f')$ *are inclusions* $i : n \to n'$ *such that* $f = f' \cdot i$

The idea is that an object (n, f) represents a multiset of n distinct elements, where the ith element $(0 \leq i \leq n-1)$ occurs $f\ i$ times. Using this structure – which entirely represents our input list's item counts – we can now map it into the semi-simplicial category Δ:

Definition B.4 *Given an occurrence function* Φ *as above, there is an induced functor* $\hat{\Phi} : \mathsf{Bag} \to \Delta$ *sending* (n, f) *to* $(\sum_i : \mathsf{Fin}\ n)\ \Phi(i, f\ i)$.

Now we can use this mapping to characterise all Φ-FEs, just as we did for k-NFEs before. A Φ-FE will be a family of permutations which are coherent to ensure filter equivariance. This is exactly a $\hat{\Phi}$-cone.

Lemma B.5 *A* Φ*-FE function is a cone over the functor* $\mathsf{Perm} \cdot \hat{\Phi}$.

What is pleasing about this construction is that we have a very similar characterisation for Φ-FEs as we had for k-NFEs. Indeed, the characterisations coincide on NFEs. Using this, we can enumerate specific mechanisms for defining filter-equivariant functions. The most interesting is the clause showing how *recursive* functions defined by *iteration* may be filter equivariant.

Lemma B.6 *The following claims are all true:*

- *If* $f \in \mathsf{FE}$ *and* $f' \in \mathsf{FE}$*, then* $f +\!\!+ f' \in \mathsf{FE}$ *and* $f \cdot f' \in \mathsf{FE}$.
- *If* $f \in \mathsf{NFE}$ *then* $f \in \mathsf{FE}$.
- *If* $\alpha : a \to [a] \to [a]$ *is such that*

$$\begin{array}{lcll} \mathsf{filter}\ \phi\ (\alpha(x, xs)) & = & \mathsf{filter}\ \phi\ xs & \phi\ x = \mathsf{false} \\ \mathsf{filter}\ \phi\ (\alpha(x, xs)) & = & \alpha(x, \mathsf{filter}\ \phi\ xs) & \phi\ x = \mathsf{true} \end{array}$$

  then $\mathsf{foldr}\ \alpha\ [] \in \mathsf{FE}$*, where* foldr *is the* right fold[6]*, such that, for a given list* $xs = [x_1, \ldots, x_n]$*:*

$$\mathsf{foldr}\ \alpha\ []\ xs = \alpha\left(x_1, \alpha\left(x_2, \alpha(\ldots, \alpha(x_{n-1}, \alpha(x_n, [])))\right)\right).$$

  Note this example covers sort, reverse *and* filter ϕ*, most of the specific FEs we discussed before.*

[5] Here $\mathsf{Fin} : \mathsf{Nat} \to \mathsf{Set}$ explicitly transforms a number to a set of that size.

[6] The general signature of the right fold is $\mathsf{foldr} : (a \to b \to b) \to b \to [a] \to b$. $\mathsf{foldr}\ f\ z : [a] \to b$ is a standard way to represent iterative computation: the function f is applied iteratively to the entries of the input list in $[a]$, with the value of $z : b$ used to "seed" the computation.