Bag Equivalence via a Proof-Relevant Membership Relation

Nils Anders Danielsson

Chalmers University of Technology and University of Gothenburg

Abstract. Two lists are *bag equivalent* if they are permutations of each other, i.e. if they contain the same elements, with the same multiplicity, but perhaps not in the same order. This paper describes how one can define bag equivalence as the presence of bijections between sets of membership proofs. This definition has some desirable properties:

– Many bag equivalences can be proved using a flexible form of equational reasoning.
– The definition generalises easily to arbitrary unary containers, including types with infinite values, such as streams.
– By using a slight variation of the definition one gets set equivalence instead, i.e. equality up to order and multiplicity. Other variations give the subset and subbag preorders.
– The definition works well in mechanised proofs.

1 Introduction

Bag (or *multiset*) *equivalence* is equality up to reordering of elements. For simplicity we can start by considering lists. The lists $[1, 2, 1]$ and $[2, 1, 1]$ are bag equivalent: $[1, 2, 1] \approx_{bag} [2, 1, 1]$. These lists are not bag equivalent to $[1, 2]$, because of differing multiplicities. *Set equivalence*, equality up to reordering and multiplicity, identifies all three lists: $[1, 2, 1] \approx_{set} [2, 1, 1] \approx_{set} [1, 2]$.

Bag equivalence is useful when specifying the correctness of certain algorithms. The most obvious example may be provided by sorting. The result of sorting something should be bag equivalent to the input: $\forall\, xs.\ sort\ xs \approx_{bag} xs$. In many cases the two sides of a bag equivalence (in this case *sort xs* and *xs*) have the same type, but this is not necessary. Consider tree sort, for instance:

$$\begin{aligned} &\textit{tree-sort} \ : \ \textit{List}\ \mathbb{N} \rightarrow \textit{List}\ \mathbb{N} \\ &\textit{tree-sort} \ = \ \textit{flatten} \circ \textit{to-search-tree} \end{aligned}$$

The function *to-search-tree* constructs binary search trees from lists, and *flatten* flattens trees. We can prove $\forall\, xs.\ \textit{tree-sort}\ xs \approx_{bag} xs$ by first establishing the following two lemmas:

$$\forall\, xs.\ \textit{to-search-tree}\ xs \approx_{bag} xs \qquad\qquad \forall\, t.\ \textit{flatten}\ t \approx_{bag} t$$

These lemmas relate trees and lists.

Another example of the utility of bag equivalence is provided by grammars. Two grammars are typically said to be equivalent if they generate the same language, i.e. the same *set* of strings. However, this is a coarse form of equivalence which identifies ambiguous and unambiguous grammars. If the languages are instead seen as *bags*, then one gets a form of equivalence which takes ambiguity into account.

Assume that *Grammar* represents grammars annotated with semantic actions, and that we have a function *parse* : *Grammar* → (*String* → *List Result*) which gives the semantics of a grammar as a function from strings to lists of results (multiple results in the case of ambiguous grammars). It is then reasonable to require that a program *opt* which transforms grammars into more optimised forms should satisfy the following property:

$$\forall\ g\ s.\ parse\ (opt\ g)\ s\ \approx_{set}\ parse\ g\ s \quad\wedge\quad parse\ (opt\ g)\ s\ \lesssim_{bag}\ parse\ g\ s$$

Here $\_\lesssim_{bag}\_$ is the *subbag preorder*: $xs\ \lesssim_{bag}\ ys$ if every element in *xs* occurs at least as often in *ys*. The property states that the new grammar should yield the same results as the old grammar ($\_\approx_{set}\_$), with no more ambiguity ($\_\lesssim_{bag}\_$). The order of the results is unspecified. Note that if we have infinitely ambiguous grammars, then the lists returned by *parse* can be infinite, in which case we need notions of set equivalence and subbag preorder adapted to such lists.

Many definitions of bag equivalence and related concepts are available in the literature, including classical definitions of permutations; definitions of bag equivalence for lists in the Coq [19], Ssreflect [7] and Coccinelle [5] libraries; and definitions of the type of bags in the Boom hierarchy [14], in terms of quotient containers [2], and in terms of combinatorial species [21, 13]. However, I want to propose another definition, based on bijections between sets of membership proofs (Sect. 3). This definition has several useful properties:

– It makes it possible to prove many equivalences using a flexible form of equational reasoning. This is demonstrated using examples in Sects. 4, 5 and 7.
– By modifying the definition slightly one gets definitions of set equivalence and the subset and subbag preorders (Sect. 8). By taking advantage of the similarity of these definitions one can avoid proof duplication: many preservation results, such as the fact that the list monad's bind operation preserves the various equivalences and preorders, can be established uniformly for all the relations with a single proof.
– The definition works for any type with a suitable membership predicate. Hoogendijk and de Moor [10] characterise a container type as a "relator" with an associated membership relation, so one might expect that the definition should work for many container types. Section 6 shows that it works for arbitrary unary containers, defined in the style of Abbott et al. [1]; this includes containers with infinite values, such as infinite streams.
– The definition works well in mechanised proofs, and has been used in practice: I used it to state and formally prove many properties of a parser combinator library [6].

Section 9 compares the definition to other definitions of bag equivalence.

To demonstrate that the definition works well in a formal setting I will use the dependently typed, functional language Agda [16, 18] below. The language is introduced briefly in Sect. 2. Code which includes all the main results in the text is, at the time of writing, available to download from my web page. (The code does not match the paper exactly. The main difference is that many definitions are universe-polymorphic, and hence a bit more general.)

2 Brief Introduction to Agda

In Agda one can define the types of *finite* (inductive) lists and unary natural numbers as follows:

```
data List (A : Set) : Set where          data ℕ : Set where
  []   : List A                            zero : ℕ
  _::_ : A → List A → List A               suc  : ℕ → ℕ
```

Here *Set* is a type of (small) types, and _::_ is an infix constructor; the underscores mark the argument positions. Values inhabiting inductive types can be destructed using structural recursion. For instance, the length of a list can be defined as follows:

```
length : {A : Set} → List A → ℕ
length []        = zero
length (x :: xs) = suc (length xs)
```

Here {*A* : *Set*} is an *implicit* argument. If Agda can infer such an argument uniquely from the context, then the argument does not need to be given explicitly, as witnessed by the recursive call to *length*. In some cases *explicit* arguments can be inferred from the context, and then one has the option of writing an underscore (_) instead of the full expression.

Types do not have to be defined using **data** declarations, they can also be defined using functions. For instance, we can define the type *Fin n*, which has exactly *n* elements, as follows:

```
Fin : ℕ → Set                       data _+_ (A B : Set) : Set where
Fin zero    = ⊥                       left  : A → A + B
Fin (suc n) = ⊤ + Fin n               right : B → A + B
```

Here ⊥ is the empty type, ⊤ the unit type (with sole inhabitant tt), and $A + B$ is the disjoint sum of the types A and B. By treating *Fin n* as a bounded number type we can define a safe lookup function:

```
lookup : {A : Set} (xs : List A) → Fin (length xs) → A
lookup []        ()
lookup (x :: xs) (left _)  = x
lookup (x :: xs) (right i) = lookup xs i
```

This function has a *dependent* type: the type of the index depends on the length of the list. The first clause contains an *absurd pattern*, (). This pattern is used to indicate to Agda that there are no values of type $Fin\ (length\ [\,])$ = Fin zero = $\bot$; note that type-checking can involve normalisation of terms, and that Agda would not have accepted this definition if we had omitted one of the cases.

Below we will use equivalences and bijections. One can introduce a type of equivalences between the types A and B using a record type as follows:

```
record _⇔_ (A B : Set) : Set where
  field to   : A → B
        from : B → A
```

To get a type of bijections we can add the requirement that the functions *to* and *from* are inverses:

```
record _↔_ (A B : Set) : Set where
  field to      : A → B
        from    : B → A
        from-to : ∀ x → from (to x) ≡ x
        to-from : ∀ x → to (from x) ≡ x
```

Here $\forall\ x \to\ \ldots$ means the same as $(x\ :\ \_) \to\ \ldots$; Agda can infer the type of x automatically.

The type $x \equiv y$ is a type of *equality proofs* showing that x and y are equal:

```
_≡_ : {A : Set} → A → A → Set
```

I take _≡_ to be the ordinary identity type of intensional Martin-Löf type theory. In particular, I do not assume that the K rule [17], which implies that all proofs of type $x \equiv y$ are equal, is available.[1] (The reason for this choice is discussed in Sect. 10.) However, for the most part it should be fine to assume that _≡_ is the usual notion of equality used in informal mathematics.

Note that _↔_ is a *dependent record type*; later fields mention earlier ones. We can use a dependent record type to define an existential quantifier (a Σ-type):

```
record ∃ {A : Set} (B : A → Set) : Set where
  constructor _,_
  field fst : A
        snd : B fst
```

A value of type $\exists\ (\lambda\ (x\ :\ A) \to B\ x)$ is a pair (x, y) containing a value x of type A and a value y of type $B\ x$. We can project from a record using the notation "record_type.field". For instance, ∃ comes with the following two projections:

```
∃.fst : {A : Set} {B : A → Set} → ∃ B → A
∃.snd : {A : Set} {B : A → Set} (p : ∃ B) → B (∃.fst p)
```

We can also use the existential quantifier to define the cartesian product of two types:

[1] By default the K rule is available in Agda, but in recent versions there is a flag that appears to turn it off.

$$
\begin{array}{l}
\_\times\_ \;:\; \mathit{Set} \to \mathit{Set} \to \mathit{Set} \\
A \times B \;=\; \exists\,(\lambda\,(\_ \;:\; A) \to B)
\end{array}
$$

The relations $\_\Leftrightarrow\_$ and $\_\leftrightarrow\_$ are equivalence relations. We can for instance prove that $\_\leftrightarrow\_$ is symmetric in the following way:

$$
\begin{array}{l}
\mathit{sym} \;:\; \{A\;B \;:\; \mathit{Set}\} \;\to\; A \;\leftrightarrow\; B \;\to\; B \;\leftrightarrow\; A \\
\mathit{sym}\;p \;=\; \textbf{record}\; \{\,\mathit{to} \;=\; \_\leftrightarrow\_.\mathit{from}\;p \;;\; \mathit{from\text{-}to} \;=\; \_\leftrightarrow\_.\mathit{to\text{-}from}\;p \\
\qquad\qquad\qquad\quad ;\; \mathit{from} \;=\; \_\leftrightarrow\_.\mathit{to}\quad p \;;\; \mathit{to\text{-}from} \;=\; \_\leftrightarrow\_.\mathit{from\text{-}to}\;p\}
\end{array}
$$

I will also use the following combinators, corresponding to reflexivity and transitivity:

$$
\begin{array}{l}
\_\square \qquad\;\;\, :\; (A \;:\; \mathit{Set}) \;\to\; A \;\leftrightarrow\; A \\
\_\leftrightarrow\langle\_\rangle\_ \;:\; (A \;:\; \mathit{Set})\;\{B\;C \;:\; \mathit{Set}\} \;\to \\
\qquad\qquad\;\; A \;\leftrightarrow\; B \;\to\; B \;\leftrightarrow\; C \;\to\; A \;\leftrightarrow\; C
\end{array}
$$

Here $\_\square$ is a unary postfix operator and $\_\leftrightarrow\langle\_\rangle\_$ a right-associative ternary mixfix operator. The choice of names and the choice of which arguments are explicit and which are implicit may appear strange, but they allow us to use a notation akin to equational reasoning for "bijectional reasoning". For instance, if we have proofs $p \;:\; A \;\leftrightarrow\; B$ and $q \;:\; C \;\leftrightarrow\; B$, then we can prove $A \;\leftrightarrow\; C$ as follows:

$$
\begin{array}{ll}
A & \leftrightarrow\langle\; p \;\rangle \\
B & \leftrightarrow\langle\; \mathit{sym}\;q \;\rangle \\
C & \square
\end{array}
$$

The idea to use mixfix operators to mimic equational reasoning notation comes from Norell [16].

To avoid clutter I will usually suppress implicit argument declarations below.

3 Bag Equivalence for Lists

For simplicity, let us start by restricting the discussion to (finite) lists. When are two lists *xs* and *ys* bag equivalent? One answer: when there is a bijection f from the positions of *xs* to the positions of *ys*, such that the value at position i in *xs* is equal to the value at position $f\;i$ in *ys*. We can formalise this as follows:

$$
\begin{array}{l}
\textbf{record}\; \_\approx'_{\mathit{bag}}\_\; (\mathit{xs}\;\mathit{ys} \;:\; \mathit{List}\;A) \;:\; \mathit{Set}\;\textbf{where} \\
\quad \textbf{field}\; \mathit{bijection} \;:\; \mathit{Fin}\;(\mathit{length}\;\mathit{xs}) \;\leftrightarrow\; \mathit{Fin}\;(\mathit{length}\;\mathit{ys}) \\
\qquad\qquad \mathit{related} \quad\;\, :\; \forall\; i \to \mathit{lookup}\;\mathit{xs}\;i \equiv \mathit{lookup}\;\mathit{ys}\;(\_\leftrightarrow\_.\mathit{to}\;\mathit{bijection}\;i)
\end{array}
$$

However, I prefer a different (but equivalent) definition.

Let us first define the *Any* predicate transformer [15]:

$$
\begin{array}{l}
\mathit{Any} \;:\; (A \to \mathit{Set}) \to \mathit{List}\;A \to \mathit{Set} \\
\mathit{Any}\;P\;[\,] \qquad\;\;\; =\; \bot \\
\mathit{Any}\;P\;(x :: \mathit{xs}) \;=\; P\;x + \mathit{Any}\;P\;\mathit{xs}
\end{array}
$$

$Any\ P\ xs$ holds if $P\ x$ holds for at least one element x of xs: $Any\ P\ [x_1, \ldots, x_n]$ reduces to $P\ x_1 + \ldots + P\ x_n + \bot$. Using Any we can define a list membership predicate:

$$\begin{aligned} &\_\in\_ \ : \ A \to List\ A \to Set \\ &x \in xs \ = \ Any\ (\lambda\ y \to x \equiv y)\ xs \end{aligned}$$

This can be read as "x is a member of xs if there is any element y of xs which is equal to x": $x \in [x_1, \ldots, x_n] \ = \ (x \equiv x_1) \ + \ \ldots \ + \ (x \equiv x_n) \ + \ \bot$. Note that $x \in xs$ is basically a subset of the positions of xs, namely those positions which contain x. Bag equivalence can then be defined as follows:

$$\begin{aligned} &\_\approx_{bag}\_ \ : \ List\ A \to List\ A \to Set \\ &xs \ \approx_{bag}\ ys \ = \ \forall\ z \ \to \ z \in xs \ \leftrightarrow \ z \in ys \end{aligned}$$

Two lists xs and ys are bag equivalent if, for any element z, the type of positions $z \in xs$ is isomorphic to (in bijective correspondence with) $z \in ys$.

It is important that $x \in xs$ can (in general) contain more than one value, i.e. that the relation is "proof-relevant". This explains the title of the paper: bag equivalence via a *proof-relevant* membership relation. If the relation were proof-*irrelevant*, i.e. if any two elements of $x \in xs$ were identified, then we would get set equivalence instead of bag equivalence.

The intuitive explanation above has a flaw. It is based on the unstated assumption that the equality type itself is proof-irrelevant: if there are several distinct proofs of $x \equiv x$, then $x \in [x]$ does *not* correspond directly to the positions of x in $[x]$. However, in the absence of the K rule the equality type is *not* necessarily proof-irrelevant [9]. Fortunately, and maybe surprisingly, one can prove that the two definitions of bag equivalence above are equivalent even in the absence of proof-irrelevance (see Sect. 5).

The first definition of bag equivalence above is, in some sense, less complicated than $\_\approx_{bag}\_$, because it does not in general involve equality of equality proofs. One may hence wonder what the point of the new, less intuitive, more complicated definition is. My main answer to this question is that $\_\approx_{bag}\_$ lends itself well to bijectional reasoning.

4 Bijectional Reasoning

How can we prove that two lists are bag equivalent? In this section I will use an example to illustrate some of the techniques that are available. The task is the following: prove that bind distributes from the left over append,

$$xs \gg\!= (\lambda\ x \to f\ x +\!\!+\ g\ x) \ \approx_{bag}\ (xs \gg\!= f) +\!\!+ (xs \gg\!= g).$$

Here bind is defined as follows:

$$\begin{aligned} &\_\gg\!=\_ \ : \ List\ A \to (A \to List\ B) \to List\ B \\ &xs \gg\!= f \ = \ concat\ (map\ f\ xs) \end{aligned}$$

The *concat* function flattens a list of lists, *map* applies a function to every element in a list, and $\_{+\!\!+}\_$ appends one list to another.

Bag equivalence is reflexive, so any equation which holds for ordinary list equality also holds for bag equivalence. To see that the equation above does not (in general) hold for ordinary list equality, let xs be $1 :: 2 :: [\,]$ and f and g both be $\lambda\ x \to x :: [\,]$, in which case the equivalence specialises as follows: $1 :: 1 :: 2 :: 2 :: [\,] \approx_{bag} 1 :: 2 :: 1 :: 2 :: [\,]$.

Before proving the left distributivity law I will introduce some basic lemmas. The first one states that *Any* is homomorphic with respect to $\_{+\!\!+}\_/\_+\_$. The lemma is proved by induction on the structure of the first list:

$$
\begin{array}{l}
\mathit{Any}\text{-}{+\!\!+}\ :\ (P\ :\ A \to \mathit{Set})\ (xs\ ys\ :\ \mathit{List}\ A) \to \\
\qquad\qquad \mathit{Any}\ P\ (xs\ {+\!\!+}\ ys)\ \leftrightarrow\ \mathit{Any}\ P\ xs + \mathit{Any}\ P\ ys \\
\mathit{Any}\text{-}{+\!\!+}\ P\ [\,]\ ys\ = \\
\quad \begin{array}{ll}
\mathit{Any}\ P\ ys & \leftrightarrow\langle\ \mathit{sym}\ {+}\text{-}\mathit{left}\text{-}\mathit{identity}\ \rangle \\
\bot + \mathit{Any}\ P\ ys & \square
\end{array} \\
\mathit{Any}\text{-}{+\!\!+}\ P\ (x :: xs)\ ys\ = \\
\quad \begin{array}{ll}
P\ x + \mathit{Any}\ P\ (xs\ {+\!\!+}\ ys) & \leftrightarrow\langle\ {+}\text{-}\mathit{cong}\ (P\ x\ \square)\ (\mathit{Any}\text{-}{+\!\!+}\ P\ xs\ ys)\ \rangle \\
P\ x + (\mathit{Any}\ P\ xs + \mathit{Any}\ P\ ys) & \leftrightarrow\langle\ {+}\text{-}\mathit{assoc}\ \rangle \\
(P\ x + \mathit{Any}\ P\ xs) + \mathit{Any}\ P\ ys & \square
\end{array}
\end{array}
$$

Note that the list xs in the recursive call $\mathit{Any}\text{-}{+\!\!+}\ P\ xs\ ys$ is structurally smaller than the input, $x :: xs$. The proof uses the following lemmas:

$$
\begin{array}{lll}
{+}\text{-}\mathit{left}\text{-}\mathit{identity} & : & \bot + A\ \leftrightarrow\ A \\
{+}\text{-}\mathit{assoc} & : & A + (B + C)\ \leftrightarrow\ (A + B) + C \\
{+}\text{-}\mathit{cong} & : & A_1\ \leftrightarrow\ A_2\ \to\ B_1\ \leftrightarrow\ B_2\ \to \\
 & & A_1 + B_1\ \leftrightarrow\ A_2 + B_2
\end{array}
$$

They state that the empty type is a left identity of $\_+\_$, and that $\_+\_$ is associative and preserves bijections. These lemmas can all be proved by defining two simple functions and proving that they are inverses.

Some readers may wonder why I did not include the step $\mathit{Any}\ P\ ([\,]\ {+\!\!+}\ ys)\ \leftrightarrow\ \mathit{Any}\ P\ ys$ in the first case of $\mathit{Any}\text{-}{+\!\!+}$. This step can be omitted because the two sides are equal *by definition*: $[\,]\ {+\!\!+}\ ys$ reduces to ys. For the same reason the step $\mathit{Any}\ P\ ((x :: xs)\ {+\!\!+}\ ys)\ \leftrightarrow\ P\ x + \mathit{Any}\ P\ (xs\ {+\!\!+}\ ys)$, which involves two reductions, can be omitted in the lemma's second case.

Note that if $\mathit{Any}\text{-}{+\!\!+}$ is applied to $\_{\equiv}\_\ z$, then we get that list membership is homomorphic with respect to $\_{+\!\!+}\_/\_+\_$: $z\ \in\ xs\ {+\!\!+}\ ys\ \leftrightarrow\ z \in xs\ +\ z \in ys$. We can use this fact to prove that $\_{+\!\!+}\_$ is commutative:

$$
\begin{array}{l}
{+\!\!+}\text{-}\mathit{comm}\ :\ (xs\ ys\ :\ \mathit{List}\ A)\ \to\ xs\ {+\!\!+}\ ys\ \approx_{bag}\ ys\ {+\!\!+}\ xs \\
{+\!\!+}\text{-}\mathit{comm}\ xs\ ys\ =\ \lambda\ z \to \\
\quad \begin{array}{ll}
z\ \in\ xs\ {+\!\!+}\ ys & \leftrightarrow\langle\ \mathit{Any}\text{-}{+\!\!+}\ (\_{\equiv}\_\ z)\ xs\ ys\ \rangle \\
z \in xs\ +\ z \in ys & \leftrightarrow\langle\ {+}\text{-}\mathit{comm}\ \rangle \\
z \in ys\ +\ z \in xs & \leftrightarrow\langle\ \mathit{sym}\ (\mathit{Any}\text{-}{+\!\!+}\ (\_{\equiv}\_\ z)\ ys\ xs)\ \rangle \\
z\ \in\ ys\ {+\!\!+}\ xs & \square
\end{array}
\end{array}
$$

$$\begin{array}{ll}
x \equiv y \;\to\; (z \equiv x) \leftrightarrow (z \equiv y) & A \times \bot \leftrightarrow \bot \\
\forall\, x \;\to\; (\exists\, \lambda\, y \to y \equiv x) \leftrightarrow \top & A \times \top \leftrightarrow A \\
B\ x \leftrightarrow (\exists\, \lambda\, y \to B\ y \times y \equiv x) & (A + B) \times C \leftrightarrow (A \times C) + (B \times C) \\
 & (\exists\, \lambda\, x \to B\ x + C\ x) \leftrightarrow \exists\, B + \exists\, C
\end{array}$$

$$\begin{array}{ll}
(\exists\, \lambda\, (i : \mathit{Fin}\ \mathsf{zero}) \to P\ i) \leftrightarrow \bot & (\exists\, \lambda\, (i : \mathit{Fin}\ (\mathsf{suc}\ n)) \to P\ i) \leftrightarrow \\
 & P\ (\mathsf{left}\ \mathsf{tt}) + (\exists\, \lambda\, (i : \mathit{Fin}\ n) \to P\ (\mathsf{right}\ i))
\end{array}$$

$$\begin{array}{ll}
A_1 \leftrightarrow A_2 \;\to\; B_1 \leftrightarrow B_2 \;\to & (\exists\, \lambda\, (x : A) \to \exists\, \lambda\, (y : B) \to C\ x\ y) \leftrightarrow \\
A_1 \times B_1 \leftrightarrow A_2 \times B_2 & (\exists\, \lambda\, (y : B) \to \exists\, \lambda\, (x : A) \to C\ x\ y)
\end{array}$$

$$(p : A_1 \leftrightarrow A_2) \;\to\; (\forall\, x \to B_1\ x \leftrightarrow B_2\ (\_{\leftrightarrow}\_.\mathit{to}\ p\ x)) \;\to\; \exists\, B_1 \leftrightarrow \exists\, B_2$$

Fig. 1. Unnamed lemmas used in proofs in Sects. 4–5 (some are consequences of others).

Here I have used the fact that _+_ is commutative: $+\text{-}\mathit{comm} : A + B \leftrightarrow B + A$. Note how commutativity of _++_ follows from commutativity of _+_.

In the remainder of the text I will conserve space and reduce clutter by not writing out the explanations within brackets, such as $\langle\ +\text{-}\mathit{comm}\ \rangle$. For completeness I list various (unnamed) lemmas used in the proofs below in Fig. 1.

Let us now consider two lemmas that relate *Any* with *concat* and *map*:

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathit{Any\text{-}concat} : (P : A \to \mathit{Set})\ (\mathit{xss} : \mathit{List}\ (\mathit{List}\ A)) \to} \\
\multicolumn{3}{l}{\qquad\qquad \mathit{Any}\ P\ (\mathit{concat}\ \mathit{xss}) \leftrightarrow \mathit{Any}\ (\mathit{Any}\ P)\ \mathit{xss}} \\
\mathit{Any\text{-}concat}\ P\ [\,] & = \bot\ \ \square & \\
\mathit{Any\text{-}concat}\ P\ (\mathit{xs} :: \mathit{xss}) & = \mathit{Any}\ P\ (\mathit{xs} +\!\!+\ \mathit{concat}\ \mathit{xss}) & \leftrightarrow \\
 & \ \mathit{Any}\ P\ \mathit{xs} + \mathit{Any}\ P\ (\mathit{concat}\ \mathit{xss}) & \leftrightarrow \\
 & \ \mathit{Any}\ P\ \mathit{xs} + \mathit{Any}\ (\mathit{Any}\ P)\ \mathit{xss} & \square
\end{array}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathit{Any\text{-}map} : (P : B \to \mathit{Set})\ (f : A \to B)\ (\mathit{xs} : \mathit{List}\ A) \to} \\
\multicolumn{3}{l}{\qquad\qquad \mathit{Any}\ P\ (\mathit{map}\ f\ \mathit{xs}) \leftrightarrow \mathit{Any}\ (P \circ f)\ \mathit{xs}} \\
\mathit{Any\text{-}map}\ P\ f\ [\,] & = \bot\ \ \square & \\
\mathit{Any\text{-}map}\ P\ f\ (x :: \mathit{xs}) & = P\ (f\ x)\quad + \mathit{Any}\ P\ (\mathit{map}\ f\ \mathit{xs}) & \leftrightarrow \\
 & \ (P \circ f)\ x + \mathit{Any}\ (P \circ f)\ \mathit{xs} & \square
\end{array}$$

Here _∘_ is function composition. If we combine *Any-concat* and *Any-map*, then we can also relate *Any* and bind:

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathit{Any\text{-}}{\gg\!=} : (P : B \to \mathit{Set})\ (\mathit{xs} : \mathit{List}\ A)\ (f : A \to \mathit{List}\ B) \to} \\
\multicolumn{3}{l}{\qquad\qquad \mathit{Any}\ P\ (\mathit{xs} \gg\!= f) \leftrightarrow \mathit{Any}\ (\mathit{Any}\ P \circ f)\ \mathit{xs}} \\
\mathit{Any\text{-}}{\gg\!=}\ P\ \mathit{xs}\ f & = \mathit{Any}\ P\ (\mathit{concat}\ (\mathit{map}\ f\ \mathit{xs})) & \leftrightarrow \\
 & \ \mathit{Any}\ (\mathit{Any}\ P)\ (\mathit{map}\ f\ \mathit{xs}) & \leftrightarrow \\
 & \ \mathit{Any}\ (\mathit{Any}\ P \circ f)\ \mathit{xs} & \square
\end{array}$$

Note that these lemmas allow us to move things between the two arguments of *Any*, the list and the predicate. When defining bag equivalence I could have defined the list membership predicate _∈_ directly, without using *Any*, but I like the flexibility which *Any* provides.

Sometimes it can be useful to switch between *Any* and $\_{\in}\_$ using the following lemma (which can be proved by induction on *xs*):

$$\mathit{Any}\text{-}{\in} \;:\; \mathit{Any}\ P\ \mathit{xs} \;\leftrightarrow\; (\exists\ \lambda\ x \rightarrow P\ x \times x \in \mathit{xs})$$

This lemma can for instance be used to show that *Any* preserves bijections and respects bag equivalence:

$$\begin{array}{l}
\mathit{Any\text{-}cong} \;:\; (P\ Q \;:\; A \rightarrow \mathit{Set})\ (\mathit{xs}\ \mathit{ys} \;:\; \mathit{List}\ A) \rightarrow \\
\qquad\qquad (\forall\ x \;\rightarrow\; P\ x \;\leftrightarrow\; Q\ x) \;\rightarrow\; \mathit{xs} \;\approx_{bag}\; \mathit{ys} \;\rightarrow \\
\qquad\qquad \mathit{Any}\ P\ \mathit{xs} \;\leftrightarrow\; \mathit{Any}\ Q\ \mathit{ys} \\
\mathit{Any\text{-}cong}\ P\ Q\ \mathit{xs}\ \mathit{ys}\ p\ \mathit{eq} \;= \\
\quad \begin{array}{ll}
\mathit{Any}\ P\ \mathit{xs} & \leftrightarrow \\
(\exists\ \lambda\ z \rightarrow P\ z \times z \in \mathit{xs}) & \leftrightarrow \\
(\exists\ \lambda\ z \rightarrow Q\ z \times z \in \mathit{ys}) & \leftrightarrow \\
\mathit{Any}\ Q\ \mathit{ys} & \Box
\end{array}
\end{array}$$

We can now prove the left distributivity law using the following non-recursive definition:

$$\begin{array}{l}
{\gg\!=}\text{-}\mathit{left\text{-}distributive} \;:\; (\mathit{xs} \;:\; \mathit{List}\ A)\ (f\ g \;:\; A \rightarrow \mathit{List}\ B) \rightarrow \\
\quad \mathit{xs} \gg\!= (\lambda\ x \rightarrow f\ x +\!\!+\ g\ x) \;\approx_{bag}\; (\mathit{xs} \gg\!= f) +\!\!+ (\mathit{xs} \gg\!= g) \\
{\gg\!=}\text{-}\mathit{left\text{-}distributive}\ \mathit{xs}\ f\ g \;=\; \lambda\ z \rightarrow \\
\quad \begin{array}{ll}
z \;\in\; \mathit{xs} \gg\!= (\lambda\ x \rightarrow f\ x +\!\!+\ g\ x) & \leftrightarrow \\
\mathit{Any}\ (\lambda\ x \rightarrow z \in f\ x +\!\!+\ g\ x)\ \mathit{xs} & \leftrightarrow \\
\mathit{Any}\ (\lambda\ x \rightarrow z \in f\ x \;+\; z \in g\ x)\ \mathit{xs} & \leftrightarrow \\
\mathit{Any}\ (\lambda\ x \rightarrow z \in f\ x)\ \mathit{xs} + \mathit{Any}\ (\lambda\ x \rightarrow z \in g\ x)\ \mathit{xs} & \leftrightarrow \\
z \in \mathit{xs} \gg\!= f \;+\; z \in \mathit{xs} \gg\!= g & \leftrightarrow \\
z \;\in\; (\mathit{xs} \gg\!= f) +\!\!+ (\mathit{xs} \gg\!= g) & \Box
\end{array}
\end{array}$$

The proof amounts to starting from both sides, using the lemmas introduced above to make the list arguments as simple as possible, and finally proving the following lemma in order to tie the two sides together in the middle:

$$\begin{array}{l}
\mathit{Any}\text{-}{+} \;:\; (P\ Q \;:\; A \rightarrow \mathit{Set})\ (\mathit{xs} \;:\; \mathit{List}\ A) \rightarrow \\
\qquad\qquad \mathit{Any}\ (\lambda\ x \rightarrow P\ x + Q\ x)\ \mathit{xs} \;\leftrightarrow\; \mathit{Any}\ P\ \mathit{xs} + \mathit{Any}\ Q\ \mathit{xs} \\
\mathit{Any}\text{-}{+}\ P\ Q\ \mathit{xs} \;= \\
\quad \begin{array}{ll}
\mathit{Any}\ (\lambda\ x \rightarrow P\ x + Q\ x)\ \mathit{xs} & \leftrightarrow \\
(\exists\ \lambda\ x \rightarrow (P\ x + Q\ x) \times x \in \mathit{xs}) & \leftrightarrow \\
(\exists\ \lambda\ x \rightarrow P\ x \times x \in \mathit{xs} + Q\ x \times x \in \mathit{xs}) & \leftrightarrow \\
(\exists\ \lambda\ x \rightarrow P\ x \times x \in \mathit{xs}) + (\exists\ \lambda\ x \rightarrow Q\ x \times x \in \mathit{xs}) & \leftrightarrow \\
\mathit{Any}\ P\ \mathit{xs} + \mathit{Any}\ Q\ \mathit{xs} & \Box
\end{array}
\end{array}$$

Note how the left distributivity property for bind is reduced to the facts that $\_{\times}\_$ and $\exists$ distribute over $\_{+}\_$ (second and third steps above).

The example above suggests that the definition of bag equivalence presented in this paper makes it possible to establish equivalences in a *modular* way, using a *flexible* form of equational reasoning: even though we are establishing a correspondence of the form $\mathit{xs} \approx_{bag} \mathit{ys}$ the reasoning need not have the form $\mathit{xs} \approx_{bag} \mathit{xs}' \approx_{bag} \ldots \approx_{bag} \mathit{ys}$.

5 The Definitions Are Equivalent

Before generalising the definition of bag equivalence I want to show that the two definitions given in Sect. 3 are equivalent.

Let us start by showing that $\_\approx_{bag}\_$ is complete with respect to $\_\approx'_{bag}\_$. We can relate the membership predicate and the lookup function as follows:

$$\textit{∈-lookup} \;:\; z \in xs \;\leftrightarrow\; \exists\,(\lambda\,(i \;:\; \mathit{Fin}\,(\mathit{length}\;xs)) \to z \equiv \mathit{lookup}\;xs\;i)$$

This lemma can be proved by induction on the list xs. It is then easy to establish completeness:

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathit{complete} \;:\; (xs\;ys \;:\; \mathit{List}\;A) \;\to\; xs \;\approx'_{bag}\; ys \;\to\; xs \;\approx_{bag}\; ys} \\
\multicolumn{2}{l}{\mathit{complete}\;xs\;ys\;eq \;=\; \lambda\,z \to} \\
\quad z \in xs & \leftrightarrow \\
\quad \exists\,(\lambda\,(i \;:\; \mathit{Fin}\,(\mathit{length}\;xs)) \to z \equiv \mathit{lookup}\;xs\;i) & \leftrightarrow \\
\quad \exists\,(\lambda\,(i \;:\; \mathit{Fin}\,(\mathit{length}\;ys)) \to z \equiv \mathit{lookup}\;ys\;i) & \leftrightarrow \\
\quad z \in ys & \Box
\end{array}$$

The second step uses the two components of eq.

Using the *∈-lookup* lemma we can also construct an isomorphism between the type of positions $\exists\;\lambda\;z \to z \in xs$ and the corresponding type of indices:

$$\begin{array}{ll}
\multicolumn{2}{l}{\textit{Fin-length} \;:\; (xs \;:\; \mathit{List}\;A) \;\to\; (\exists\;\lambda\;z \to z \in xs) \;\leftrightarrow\; \mathit{Fin}\;(\mathit{length}\;xs)} \\
\multicolumn{2}{l}{\textit{Fin-length}\;xs \;=} \\
\quad (\exists\;\lambda\;z \to z \in xs) & \leftrightarrow \\
\quad (\exists\;\lambda\;z \to \exists\;\lambda\,(i \;:\; \mathit{Fin}\,(\mathit{length}\;xs)) \to z \equiv \mathit{lookup}\;xs\;i) & \leftrightarrow \\
\quad (\exists\;\lambda\,(i \;:\; \mathit{Fin}\,(\mathit{length}\;xs)) \to \exists\;\lambda\;z \to z \equiv \mathit{lookup}\;xs\;i) & \leftrightarrow \\
\quad \mathit{Fin}\;(\mathit{length}\;xs) \times \top & \leftrightarrow \\
\quad \mathit{Fin}\;(\mathit{length}\;xs) & \Box
\end{array}$$

The penultimate step uses the fact that, for any x, types of the form $\exists\;\lambda\;y \to y \equiv x$ are "contractible" [20, Lemma `idisweq`], and hence isomorphic to the unit type. One can easily reduce this fact to the problem of proving that (x, refl) is equal to (y, eq), for arbitrary y and $eq \;:\; y \equiv x$, where $\mathsf{refl} \;:\; \{A \;:\; \mathit{Set}\}\;\{z \;:\; A\} \to z \equiv z$ is the canonical proof of reflexivity. This follows from a single application of the J rule—the usual eliminator for the Martin-Löf identity type—which in this case allows us to pattern match on eq, replacing it with refl and unifying y and x.

As an aside one can note that *Fin-length* is a generalisation of the fact above (this observation is due to Thierry Coquand). The statement of *Fin-length* may be a bit more suggestive if the existential is written as a Σ-type:

$$(\Sigma\;x \;:\; A.\; x \equiv x_1 + \ldots + x \equiv x_n) \;\leftrightarrow\; \mathit{Fin}\;n.$$

Note that this statement is proved without assuming that the equality type is proof-irrelevant. We can for instance instantiate A with the universe *Set* and all the x_i with the type $\mathbb{N}$.[2] In homotopy type theory [20] there are infinitely many distinct proofs of $\mathbb{N} \equiv \mathbb{N}$, but *Fin-length* is still valid.

We can use *Fin-length* to construct an index bijection from a bag equivalence:

$$
\begin{array}{l}
\textit{Fin-length-cong} \;:\; (\textit{xs}\ \textit{ys} \;:\; \textit{List}\ A) \;\to\; \textit{xs} \approx_{bag} \textit{ys} \;\to \\
\qquad\qquad\qquad\quad \textit{Fin}\ (\textit{length}\ \textit{xs}) \leftrightarrow \textit{Fin}\ (\textit{length}\ \textit{ys}) \\
\textit{Fin-length-cong}\ \textit{xs}\ \textit{ys}\ \textit{eq} \;= \\
\quad \textit{Fin}\ (\textit{length}\ \textit{xs}) \quad \leftrightarrow \\
\quad \exists\ (\lambda\ z \to z \in \textit{xs}) \quad \leftrightarrow \\
\quad \exists\ (\lambda\ z \to z \in \textit{ys}) \quad \leftrightarrow \\
\quad \textit{Fin}\ (\textit{length}\ \textit{ys}) \quad \Box
\end{array}
$$

All that remains in order to establish soundness of $\_\approx_{bag}\_$ with respect to $\_\approx'_{bag}\_$ is to show that the positions which the bijection *Fin-length-cong xs ys eq* relates contain equal elements. This bijection is defined using a number of lemmas which I have postulated above. If these lemmas are instantiated with concrete definitions in a suitable way (as in the code which accompanies the paper), then the result can be established using a short proof. Thus we get soundness:

$$\textit{sound} \;:\; (\textit{xs}\ \textit{ys} \;:\; \textit{List}\ A) \;\to\; \textit{xs} \approx_{bag} \textit{ys} \;\to\; \textit{xs} \approx'_{bag} \textit{ys}$$

6 Bag Equivalence for Arbitrary Containers

The definition of bag equivalence given in Sect. 3 generalises from lists to many other types. Whenever we can define the *Any* type we get a corresponding notion of bag equivalence. The definition is not limited to types with finite values. We can for instance define *Any* for infinite streams (but in that case *Any* can not be defined by structural recursion as in Sect. 3).

It turns out that *containers*, in the style of Abbott et al. [1], make it very easy to define *Any*. The unary containers which I will present below can be used to represent arbitrary strictly positive simple types in one variable (in a certain extensional type theory [1]), so we get a definition of bag equivalence which works for a very large set of types. By using n-ary containers, or indexed containers [4], it should be possible to handle even more types, but I fear that the extra complexity would obscure the main idea, so I stick to unary containers here.

A (unary) container consists of a type of shapes and, for every shape, a type of positions:

$$
\begin{array}{l}
\textbf{record}\ \textit{Container} \;:\; \textit{Set}_1\ \textbf{where} \\
\quad \textbf{constructor}\ \_\rhd\_ \\
\quad \textbf{field}\ \textit{Shape} \quad\ \;:\; \textit{Set} \\
\qquad\qquad \textit{Position} \;:\; \textit{Shape} \to \textit{Set}
\end{array}
\qquad
\begin{array}{l}
[\![\_]\!] \;:\; \textit{Container} \to \textit{Set} \to \textit{Set} \\
[\![\ S \rhd P\ ]\!]\ A \;= \\
\quad \exists\ \lambda\ (s \;:\; S) \;\to\; (P\ s \to A)
\end{array}
$$

[2] After making the definitions universe-polymorphic; see the accompanying code.

(Set_1 is a type of large types.) A container C can be interpreted as a type constructor $[\![\ C\ ]\!]$. Values of type $[\![\ S \rhd P\ ]\!]\ A$ have the form (s, f), where s is a shape and f is a function mapping the positions corresponding to s to values.

Let us take some examples:

- We can represent finite lists using $\mathbb{N} \rhd \mathit{Fin}$: the shape is the length of the list, and a list of length n has n positions.
- Infinite streams can be represented as follows: $\top \rhd (\lambda\ \_ \to \mathbb{N})$. There is only one shape, and this shape comes with infinitely many positions.
- Consider finite binary trees with values in the internal nodes:

  $$\begin{array}{l} \textbf{data}\ \mathit{Tree}\ (A\ :\ \mathit{Set})\ :\ \mathit{Set}\ \textbf{where} \\ \quad \mathsf{leaf}\ \ \ :\ \mathit{Tree}\ A \\ \quad \mathsf{node}\ :\ \mathit{Tree}\ A \to A \to \mathit{Tree}\ A \to \mathit{Tree}\ A \end{array}$$

  This type can be represented by $S \rhd P$, where S and P are defined as follows (note that Agda supports overloaded constructors):

  $$\begin{array}{l} \textbf{data}\ S\ :\ \mathit{Set}\ \textbf{where} \\ \quad \mathsf{leaf}\ \ \ :\ S \\ \quad \mathsf{node}\ :\ S \to S \to S \end{array} \qquad \begin{array}{lcl} P\ :\ S \to \mathit{Set} & & \\ P\ \mathsf{leaf} & = & \bot \\ P\ (\mathsf{node}\ l\ r) & = & P\ l + \top + P\ r \end{array}$$

  The shapes are unlabelled finite binary trees, and the positions are paths to the internal nodes.

Note that the type of shapes can be obtained by applying the container's type constructor to the unit type. For instance, S is isomorphic to $\mathit{Tree}\ \top$.

Given a container we can define Any as follows [3] (where I have written out the implicit argument $\{S \rhd P\}$ in order to be able to give a type signature for p):

$$\begin{array}{l} \mathit{Any}\ :\ \{C\ :\ \mathit{Container}\}\ \{A\ :\ \mathit{Set}\} \to (A \to \mathit{Set}) \to ([\![\ C\ ]\!]\ A \to \mathit{Set}) \\ \mathit{Any}\ \{S \rhd P\}\ Q\ (s, f)\ =\ \exists\ \lambda\ (p\ :\ P\ s) \to Q\ (f\ p) \end{array}$$

$\mathit{Any}\ Q\ (s, f)$ consists of pairs (p, q) where p is an s-indexed position and q is a proof showing that the value at position p satisfies the predicate Q.

We can now define bag equivalence as before. In fact, we can define bag equivalence for values of *different* container types, as long as the elements they contain have the same type:

$$\begin{array}{l} \_\in\_\ :\ A \to [\![\ C\ ]\!]\ A \to \mathit{Set} \\ x \in \mathit{xs}\ =\ \mathit{Any}\ (\lambda\ y \to x \equiv y)\ \mathit{xs} \end{array} \qquad \begin{array}{l} \_\approx_{bag}\_\ :\ [\![\ C_1\ ]\!]\ A \to [\![\ C_2\ ]\!]\ A \to \mathit{Set} \\ \mathit{xs}\ \approx_{bag}\ \mathit{ys}\ = \\ \quad \forall\ z\ \ \to\ \ z \in \mathit{xs}\ \leftrightarrow\ z \in \mathit{ys} \end{array}$$

We can also generalise the alternative definition $\_\approx'_{bag}\_$ from Sect. 3:

$$\begin{array}{l} \_\approx'_{bag}\_\ :\ \{C_1\ C_2\ :\ \mathit{Container}\}\ \{A\ :\ \mathit{Set}\} \to [\![\ C_1\ ]\!]\ A \to [\![\ C_2\ ]\!]\ A \to \mathit{Set} \\ \_\approx'_{bag}\_\ \{S_1 \rhd P_1\}\ \{S_2 \rhd P_2\}\ (s_1, f_1)\ (s_2, f_2)\ = \\ \quad \exists\ \lambda\ (b\ :\ P_1\ s_1\ \leftrightarrow\ P_2\ s_2)\ \ \to\ \ \forall\ p \to f_1\ p \equiv f_2\ (\_\leftrightarrow\_.\mathit{to}\ b\ p) \end{array}$$

This definition states that two values are bag equivalent if there is a bijection between their positions which relates equal elements. As before $\_\approx_{bag}\_$ and $\_\approx'_{bag}\_$ are equivalent. The proof is easier than the one in Sect. 5: the generalisation of $\in$-*lookup* holds by definition.

7 More Bijectional Reasoning

Let us now revisit the tree sort example from the introduction. To avoid minor complications related to the container encoding I use the direct definition of the *Tree* type from Sect. 6, and define *Any* and membership explicitly:

```
AnyTree : (A → Set) → (Tree A → Set)
AnyTree P leaf           = ⊥
AnyTree P (node l x r) =
  AnyTree P l ⊎ P x ⊎ AnyTree P r
```

```
_∈Tree_ : A → Tree A → Set
x ∈Tree t =
  AnyTree (λ y → x ≡ y) t
```

The *flatten* function can be defined (inefficiently) as follows:

```
flatten : Tree A → List A
flatten leaf           = []
flatten (node l x r) = flatten l ++ x :: flatten r
```

The *flatten* lemma from the introduction can then be proved as follows (where $\_\in_{List}\_$ refers to the definition of list membership from Sect. 3):

```
flatten-lemma : (t : Tree A) → ∀ z → z ∈List flatten t ↔ z ∈Tree t
flatten-lemma leaf           = λ z → ⊥ □
flatten-lemma (node l x r) = λ z →
  z ∈List flatten l ++ x :: flatten r            ↔
  z ∈List flatten l ⊎ z ≡ x ⊎ z ∈List flatten r  ↔
  z ∈Tree l         ⊎ z ≡ x ⊎ z ∈Tree r          □
```

In the leaf case the two sides evaluate to the empty type. The node case contains two steps: the first one uses *Any*-++, and the second one uses the inductive hypothesis twice.

With a suitable definition of *to-search-tree* it is not much harder to prove the following lemma (see the accompanying code):

```
to-search-tree-lemma :
  (xs : List ℕ) → ∀ z → z ∈Tree to-search-tree xs ↔ z ∈List xs
```

It is then easy to prove that *tree-sort* produces a permutation of its input:

```
tree-sort-permutes : (xs : List ℕ) → tree-sort xs ≈bag xs
tree-sort-permutes xs = λ z →
  z ∈List flatten (to-search-tree xs)  ↔
  z ∈Tree to-search-tree xs            ↔
  z ∈List xs                           □
```

8 Set Equivalence, Subsets and Subbags

It is easy to tweak the definition of bag equivalence so that we get set equivalence:

$$\begin{array}{l} \_{\approx_{set}}\_ \ : \ \mathit{List}\ A \rightarrow \mathit{List}\ A \rightarrow \mathit{Set} \\ \mathit{xs}\ \approx_{set}\ \mathit{ys}\ =\ \forall\ z\ \ \rightarrow\ \ z \in \mathit{xs} \Leftrightarrow z \in \mathit{ys} \end{array}$$

This definition states that xs and ys are set equivalent if, for any value z, z is a member of xs iff it is a member of ys. We can also define subset and subbag relations:

$$\begin{array}{ll} \_{\lesssim_{set}}\_ \ : \ \mathit{List}\ A \rightarrow \mathit{List}\ A \rightarrow \mathit{Set} & \_{\lesssim_{bag}}\_ \ : \ \mathit{List}\ A \rightarrow \mathit{List}\ A \rightarrow \mathit{Set} \\ \mathit{xs}\ \lesssim_{set}\ \mathit{ys}\ = & \mathit{xs}\ \lesssim_{bag}\ \mathit{ys}\ = \\ \quad \forall\ z\ \ \rightarrow\ \ z \in \mathit{xs} \rightarrow z \in \mathit{ys} & \quad \forall\ z\ \ \rightarrow\ \ z \in \mathit{xs} \rightarrowtail z \in \mathit{ys} \end{array}$$

Here $A \rightarrowtail B$ stands for the type of injections from A to B: xs is a subbag of ys if every element occurs at least as often in ys as in xs.

It is now easy to generalise over the kind of function space used in the four definitions and define $xs \sim[\ k\ ]\ ys$, meaning that xs and ys are k-related, where k ranges over subset, set, subbag and bag. Using this definition one can prove many preservation properties uniformly for all four relations at once (given suitable combinators, some of which may not be defined uniformly). Here is one example of such a preservation property:

$$\begin{array}{l} {\gg\!=}\text{-}\mathit{cong}\ :\ (\mathit{xs}\ \mathit{ys}\ :\ \mathit{List}\ A)\ (f\ g\ :\ A \rightarrow \mathit{List}\ B) \rightarrow \\ \qquad\qquad \mathit{xs} \sim[\ k\ ]\ \mathit{ys} \rightarrow (\forall\ x \rightarrow f\ x \sim[\ k\ ]\ g\ x) \rightarrow \mathit{xs} \gg\!= f \sim[\ k\ ]\ \mathit{ys} \gg\!= g \end{array}$$

Details of these constructions are not provided in the paper due to lack of space. See the accompanying code for more information.

9 Related Work

Morris [15] defines *Any* for arbitrary indexed strictly positive types. The dual of *Any*, *All*, goes back at least to Hermida and Jacobs [8], who define it for polynomial functors. In Hoogendijk and de Moor's treatment of containers [10] membership is a lax natural transformation, and this implies that the following variant of *Any-map* (with $\_\Leftrightarrow\_$ rather than $\_\leftrightarrow\_$) holds: $x \in \mathit{map}\ f\ \mathit{ys} \Leftrightarrow \exists\ \lambda\ y \rightarrow x \equiv f\ y\ \times\ y \in \mathit{ys}$.

In a previous paper I used the definitions of bag and set equivalence given above in order to state and formally prove properties of a parser combinator library [6]. That paper did not discuss bijectional reasoning, did not discuss alternative definitions of bag equivalence such as $\_{\approx'_{bag}}\_$, and did not define bag and set equivalence for arbitrary containers, so the overlap with the present paper is very small. The paper did define something resembling bag and set equivalence for parsers. Given that $x \in p \cdot s$ means that x is one possible result of applying the parser p to the string s we can define the relations as follows:

$p_1 \approx p_2 \; = \; \forall \, x \; s \;\; \to \;\; x \in p_1 \cdot s \;\; \sim \;\; x \in p_2 \cdot s$. When $\sim$ is $\Leftrightarrow$ we get *language equivalence*, and when it is $\leftrightarrow$ we get the stronger notion of *parser equivalence*, which distinguishes parsers that exhibit differing amounts of ambiguity. Correctness of the *parse* function, which takes a parser and an input string to a list of results, was stated as follows: $x \in p \cdot s \;\; \leftrightarrow \;\; x \in \mathit{parse} \; p \; s$. Notice the flexibility provided by the use of bijections: the two sides of the correctness statement refer to different things—an inductive definition of the semantics of parsers to the left, and list membership to the right—and yet they can be usefully related.

Abbott et al. [2] define bags using *quotient containers*. A quotient container is a container $S \rhd P$ plus, for each shape s, a set $G \; s$ of automorphisms on $P \; s$, containing the identity and closed under composition and inverse. Quotient containers are interpreted as ordinary containers, except that the position-to-value functions of type $P \; s \to A$ (for some A) are quotiented by the equivalence relation that identifies f_1 and f_2 if $f_2 \; = \; f_1 \circ g$ for some $g \; : \; G \; s$. Abbott et al. define bags by taking the list container $\mathbb{N} \rhd \mathit{Fin}$ and letting $G \; n$ be the symmetric group on $\mathit{Fin} \; n$: $G \; n \; = \; \mathit{Fin} \; n \;\; \leftrightarrow \;\; \mathit{Fin} \; n$. The position-to-value functions of $\mathbb{N} \rhd \mathit{Fin}$ correspond to the *lookup* function, so this definition of bags is very close to what you get if you quotient lists by $\_{\approx'_{bag}}\_$, the alternative definition of bag equivalence given in Sect. 3. Quotient containers only allow us to identify values which have the same shape, so one could not define bags by starting from the binary tree container defined in Sect. 6 and turning this into a quotient container, at least not in an obvious way.

In the SSReflect [7] library bag equivalence (for finite lists containing elements with decidable equality) is defined as a boolean-valued computable function: the list xs is a permutation of ys if, for every element z of $xs \mathbin{+\!\!+} ys$, the number of occurrences of z in xs is equal to the number of occurrences in ys.

The Coq [19] standard library contains (at least) two definitions related to bag equivalence. A multiset containing values of type A, where A comes with decidable equality, is defined as a function of type $A \to \mathbb{N}$, i.e. as a function associating a multiplicity with every element. There is also an inductive definition of bag equivalence which states (more or less) that xs and ys are bag equivalent if xs can be transformed into ys using a finite sequence of transpositions of adjacent elements. It is easy to tweak this definition to get set equivalence, but it does not seem easy to generalise it to arbitrary containers.

Contejean [5] defines bag equivalence for lists inductively by, in effect, enumerating where every element in the left list occurs in the right one. It seems likely that this definition can be adapted to streams, but it is not obvious how to generalise it to branching structures such as binary trees.

In the Boom hierarchy (attributed to Boom by Meertens [14]) the type of bags containing elements of type A is defined as the free commutative monoid on A, i.e. bags are lists where the append operation is taken to be commutative. The type of sets is defined by adding the requirement that the append operation is idempotent. Generalising to types with infinite values seems nontrivial. Hoogendijk [11] and Hoogendijk and Backhouse [12], working with the Boom hierarchy in a relational setting, prove various laws related to bags and sets (as well as lists and binary trees). One result is that the map function preserves bag and set equivalence.

Yorgey [21] points out that one can define the type of bags as a certain (finitary) combinatorial species [13]. A species is an endofunctor in the category of finite sets and bijections; one can see the endofunctor as mapping a set of position labels to a labelled structure. Bags correspond to the species which maps a set A to the singleton set $\{ A \}$, and lifts a bijection $A \leftrightarrow B$ in the obvious way.

10 Conclusions

Through a number of examples, proofs and generalisations I hope to have shown that the definition of bag equivalence presented in this paper is useful. I do not claim that this definition is always preferable to others. For instance, in the absence of proof-irrelevance it seems to be easier to prove that cons is left cancellative using the definition $\_\approx'_{bag}\_$ from Sect. 3 (see the accompanying code). However, $\_\approx_{bag}\_$ and $\_\approx'_{bag}\_$ are equivalent, so in many cases it should be possible to use one definition in one proof and another in another.

As mentioned above I have been careful not to use the K rule when formalising this work. The reason is the ongoing work on homotopy type theory [20], a form of type theory where equality of types is (equivalent to) isomorphism and equality of functions is pointwise equality. With this kind of type theory bag equivalence can be stated as $xs \;\approx_{bag}\; ys \;=\; (\lambda\; z \rightarrow z \in xs) \equiv (\lambda\; z \rightarrow z \in ys)$, the bijectional reasoning in this paper can be turned into equational reasoning, and preservation lemmas like $+$-*cong* do not need to be proved (because equality is substitutive). However, homotopy type theory is incompatible with the K rule, which implies that all proofs of $A \equiv B$ are equal: the equalities corresponding to the identity function and the not function should be distinct elements of $Bool \equiv Bool$.

Acknowledgements. I would like to thank Simon Thompson and Bengt Nordström for suggesting that my definition of bag and set equivalence could be generalised further, to include other relations; and Fritz Henglein for suggesting that I should use the term "bag equivalence" rather than "bag equality". I would also like to thank Thierry Coquand for suggesting that *Any* could be defined as an iterated sum rather than as an inductive family, and for urging me to make sure that the results are valid also in the absence of the K rule. Finally I would like to thank several anonymous reviewers for useful feedback. In particular, one reviewer suggested that I could use tree sort as an example.

Large parts of this work were done when I was working at the University of Nottingham, with financial support from EPSRC (grant code: EP/E04350X/1). I have also received support from the ERC: "The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement n° 247219."

References

[1] Abbott, M., Altenkirch, T., Ghani, N.: Containers: Constructing strictly positive types. Theoretical Computer Science 342, 3–27 (2005)

[2] Abbott, M., Altenkirch, T., Ghani, N., McBride, C.: Constructing polymorphic programs with quotient types. In: Kozen, D. (ed.) MPC 2004. LNCS, vol. 3125, pp. 2–15. Springer, Heidelberg (2004)

[3] Altenkirch, T., Levy, P., Staton, S.: Higher-order containers. In: Ferreira, F., Löwe, B., Mayordomo, E., Mendes Gomes, L. (eds.) CiE 2010. LNCS, vol. 6158, pp. 11–20. Springer, Heidelberg (2010)

[4] Altenkirch, T., Morris, P.: Indexed containers. In: LICS '09. pp. 277–285 (2009)

[5] Contejean, E.: Modeling permutations in Coq for Coccinelle. In: Comon-Lundh, H., Kirchner, C., Kirchner, H. (eds.) Rewriting, Computation and Proof, LNCS, vol. 4600, pp. 259–269. Springer, Heidelberg (2007)

[6] Danielsson, N.A.: Total parser combinators. In: ICFP'10. pp. 285–296 (2010)

[7] Gonthier, G., Mahboubi, A., Tassi, E.: A small scale reflection extension for the Coq system. Tech. Rep. inria-00258384, version 10, INRIA (2011)

[8] Hermida, C., Jacobs, B.: An algebraic view of structural induction. In: Pacholski, L., Tiuryn, J. (eds.) CSL '94. LNCS, vol. 933, pp. 412–426. Springer, Heidelberg (1995)

[9] Hofmann, M., Streicher, T.: The groupoid model refutes uniqueness of identity proofs. In: LICS '94. pp. 208–212 (1994)

[10] Hoogendijk, P., de Moor, O.: Container types categorically. Journal of Functional Programming 10(2), 191–225 (2000)

[11] Hoogendijk, P.F.: (Relational) programming laws in the Boom hierarchy of types. In: Bird, R.S., Morgan, C.C., Woodcock, J.C.P. (eds.) Mathematics of Program Construction, Second International Conference. LNCS, vol. 669, pp. 163–190. Springer, Heidelberg (1993)

[12] Hoogendijk, P.F., Backhouse, R.C.: Relational programming laws in the tree, list, bag, set hierarchy. Science of Computer Programming 22(1–2), 67–105 (1994)

[13] Joyal, A.: Une théorie combinatoire des séries formelles. Advances in Mathematics 42(1), 1–82 (1981)

[14] Meertens, L.: Algorithmics: Towards programming as a mathematical activity. In: Mathematics and Computer Science, CWI Monographs, vol. 1, pp. 289–334. North-Holland (1986)

[15] Morris, P.W.J.: Constructing Universes for Generic Programming. Ph.D. thesis, The University of Nottingham (2007)

[16] Norell, U.: Towards a practical programming language based on dependent type theory. Ph.D. thesis, Chalmers University of Technology and Göteborg University (2007)

[17] Streicher, T.: Investigations Into Intensional Type Theory. Habilitationsschrift, Ludwig-Maximilians-Universität München (1993)

[18] The Agda Team: The Agda Wiki. Available at `http://wiki.portal.chalmers.se/agda/` (2012)

[19] The Coq Development Team: The Coq Proof Assistant Reference Manual, Version 8.3pl3 (2011)

[20] Voevodsky, V.: Univalent foundations project (a modified version of an NSF grant application) (2010), unpublished

[21] Yorgey, B.A.: Species and functors and types, oh my! In: Haskell'10. pp. 147–158 (2010)