What's in a Bag?

An "Application Proving Interface" for Finite Bags and its Implementation

Alexander Dinges
Ralf Hinze
alexander.dinges@cs.rptu.de
ralf.hinze@cs.rptu.de
RPTU Kaiserslautern-Landau
Kaiserslautern, Germany

ABSTRACT

Bags are ubiquitous in program verification. They are the means of choice when we want to express that a collection of elements is a rearrangement of another collection. We are working towards an "application proving interface" (API) for finite bags that is perspicuous, rich, and easy to use. We propose an implementation of the Bag API in the dependently typed language Agda that has minimal meta-theoretic requirements and that we believe is suitable for both instructional and industrial applications. Bags form a free commutative monoid. The implementation boils down to the free structure: bag expressions built from the empty bag ⟅⟆, singleton bags ⟅ x ⟆, and the union of bags $A \uplus B$, quotiented by the laws of commutative monoids.

ACM Reference Format:
Alexander Dinges and Ralf Hinze. 2023. What's in a Bag?: An "Application Proving Interface" for Finite Bags and its Implementation. In *Proceedings of ACM Conference (IFL'23).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Bags are an important part of the program verification toolbox. A bag is a collection of elements that takes account of their multiplicity but not of their order. Consequently, bags are the means of choice when we want to express that a collection of elements is a *rearrangement* of another collection. Typical examples are representation changers: Sorting a list; constructing a heap from a list; flattening a tree to a list; converting between binary trees and forests of multiway trees; etc. In each case, we want to make sure that no elements are lost, no elements are duplicated or invented. Bags also play an important role in the specification of abstract datatypes: Looking up a key in a search tree is successful if and only if the key has been inserted beforehand; every element extracted from a priority queue has been added beforehand; etc.

Desiderata. Before we can use bags, we have to define them. However, unlike lists, finite bags do not enjoy a simple inductive definition. Indeed, the design space for implementing bags is surprisingly large, especially in a dependently typed setting such as Agda [4]. Before we get an overview of the different options, let us first establish some guiding criteria for the design of a Bag API and its implementation, which we are working towards in this paper.

- *Clarity:* Are interface and implementation perspicuous? Are they suitable for use in a first course on program verification (which deals with the correctness of sorting algorithms as a first non-trivial example)?
- *Requirements:* What are the requirements on the meta-theory? Are there any assumptions on the underlying element type, such as decidable equality or total ordering?
- *Ease of use:* Can Agda's proof synthesizer *Agsy* discharge simple proof obligations? How easy is it to define functions over bags?
- *Completeness:* Is the API sufficiently rich for practical applications? Are "set-like" operations available? What about important properties such as cancellation? Is there support for different recursion patterns, say, induction over the cardinality of a bag?

The criteria are, of course, somewhat subjective. In particular, they are geared towards didactic applications.

Related work. The existing approaches can be roughly divided into four categories.

- *Ad-hoc approaches:* Some solve specialised problems, for example: When is a list a permutation of another list? See Figure 1, which is a minor rewrite of van Laarhoven's blog post. Cunning as it is, this approach is *too concrete* and *too specific* — elementary properties such as transitivity of $\_\approx\_$ require considerable proof effort.
- *Approaches based on multiplicity:* Putting the math spectacles on, bags can be represented as finite maps into the naturals, $X \to_{\text{fin}} \mathbb{N}$, mapping each potential element to its multiplicity. This approach is *too restrictive* as it requires decidable equality — consider defining singleton bags.
- *Approaches based on sequence types:* Here is a general recipe for constructing bag types. Pick an arbitrary sequence type and endow it with an equivalence relation that abstracts away from the order of elements. This spans a 2-dimensional design space. Choices for sequence types include
  - *finite maps from positions to elements, Fin $n \to X$*;
  - *standard lists*; or
  - *join lists*, which are akin to bag expressions built from the empty bag ⟅⟆, singletons ⟅ x ⟆, and union $A \uplus B$.

  There are, at least, four options for defining the underlying equivalence relation:
  - Using *permutations* [5]: Two sequences are equivalent iff a bijection on positions can be exhibited.
  - Via *proof-relevant membership* [8]: Two sequences A and B are equivalent iff for each proof of $x \in A$ there is a corresponding proof $x \in B$, and vice versa. This is *too*

IFL'23, August 2023, Braga, Portugal
2023. https://doi.org/10.1145/nnnnnnn.nnnnnnn

restrictive as it is confined to element types with *propositional* equality (or, at least, unique identity proofs).
  - Using *multiplicity*: Two sequences are equivalent iff the multiplicities of each element are equal. Again, this is *too restrictive* as it requires decidable equality.
  - Using *proof trees*: Two bag expressions are equivalent iff there is a proof using the laws of commutative monoids.
- *Approaches based on quotient types:* Bags can finally be defined as a higher inductive type (HIT). The options are, in principle, similar to the previous approach. Take an arbitrary, inductively defined sequence type and endow it with identities that abstract away from order. While this is an exciting line of research, HITs are probably *too sophisticated* for a first course on program verification.

The set of all finite bags forms a *free* commutative monoid. Our implementation amounts to the free structure: bag expressions quotiented by the laws of commutative monoids. This is, perhaps, the simplest, most straightforward implementation one can think of. Rather surprisingly, it turned out to be the most practical in terms of proof effort and convenience of use.

Contributions. Our paper makes the following contributions:

- we design a rich Bag API that includes "set-like" operations, properties such as cancellation, and recursion schemes;
- we provide an implementation of the Bag API that is perspicuous, easy (even fun) to use, and that we believe is suitable for both instructional and industrial applications;
- the implementation has minimal meta-theoretic requirements and does not impose any restrictions on the element type (unless they are unavoidable for principled reasons);
- the implementation serves nicely as a blueprint for other free structures: lists, the free monoid, and finite sets, the free bounded semilattice.

Overview. The remainder of the paper is structured as follows. Section 2 introduces the "application proving interface" (API) with examples. Sections 3–8 detail the implementation of the Bag API. Along the way we discuss alternative approaches. Section 9 reviews related work in more depth and, finally, Section 10 concludes.

The paper is aimed at a reader who is familiar with the basics of Agda and and program verification, say, to the level of Stump [12].

2 USE OF BAGS IN PROGRAM VERIFICATION

In this section, the Bag API is introduced by means of examples. As a first concrete application, consider a sorting function, say, *sorting by insertion*:

```
sort : List Elem → List Elem
sort []        = []
sort (a :: as) = insert a (sort as)
insert : Elem → List Elem → List Elem
insert a [] = a :: []
insert a (b :: bs) with a ⩽ ? ⩾′ b
... | LE    = a :: b :: bs
... | GE    = b :: insert a bs
```

A sorting function has to satisfy two correctness properties:

(1) The output is ordered.
(2) The output is a permutation of the input.

The first property often attracts a lot of attention, whereas the second is woefully neglected. Clearly, either property alone is insufficient: the constant function that always returns the empty list satisfies the first, the identity function the second property.

One can only speculate about the reasons why the permutation property gets less attention. Perhaps, because it is so "obvious": inspecting, say, the second equation of *sort*, we "see" that each variable that is introduced on the left-hand side appears exactly once on the right-hand side — but, of course, "visual" code inspection can be misleading. Perhaps, because formal proofs of the property are cumbersome, especially if we actually try to exhibit the underlying permutation, the function that maps positions of input elements to positions in the output.

Bags come to the rescue. Since bags abstract away from the order of elements, a list is a permutation of another list if and only if they are equivalent as bags. So as a preparatory step towards showing the second correctness property, we define a function that turns a list into a bag:

```
bag : List Elem → Bag Elem
bag []        = ≀∫
bag (a :: as) = ≀ a ∫ ⊎ bag as
```

The function *bag* illustrates the three principled ways of forming a bag: (1) ≀∫ is the empty bag, which contains no elements; (2) ≀ x ∫ denotes the bag that contains a single occurrence of x; (3) A ⊎ B denotes the bag union of A and B, also known as the sum of A and B as the multiplicities of elements are added.

A fairly straightforward, inductive proof establishes the permutation property of sorting by insertion:

```
sort~ : ∀ as → bag (sort as) ~ bag as
sort~ [] =
  proof
    bag (sort [])
  ~⟨ ι ⟩
    bag []
  ∎
sort~ (a :: as) =
  proof
    bag (sort (a :: as))
  ~⟨ ι ⟩
    bag (insert a (sort as))
  ~⟨ insert~ a (sort as) ⟩
    ≀ a ∫ ⊎ bag (sort as)
  ~⟨ ι ⊎ sort~ as ⟩
    ≀ a ∫ ⊎ bag as
  ~⟨ ι ⟩
    bag (a :: as)
  ∎
```

The equality of bags is written $A \sim B$, pronounced "A equivales B". The proof above uses a format usually attributed to Wim Feijen: the term p in A ~⟨ p ⟩ B *evidences* the equivalence of A and B. For example, reflexivity is evidenced by ι like identity. Given proofs

```
select(A, As, [A|As]).
select(A, [B|As], [B|Bs]) :-
    select(A, As, Bs).

permutation([], []).
permutation([A|As], Bs) :-
    select(A, Xs, Bs),
    permutation(As, Xs).
```

```
data Select : A → List A → List A → Set where
  head : Select a as (a ∷ as)
  tail  : Select a as bs → Select a (b ∷ as) (b ∷ bs)
data _≈_ : List A → List A → Set where
  []    : [] ≈ []
  _∷_ : Select a xs bs → as ≈ xs → a ∷ as ≈ bs
```

Figure 1: When is a list a permutation of another list, a Prolog-inspired approach.

$p : A_1 \sim A_2$ and $q : B_1 \sim B_2$, the term $p \uplus q$ demonstrates the equivalence of $A_1 \uplus B_1$ and $A_2 \uplus B_2$.

The proof format is targeted at the human reader. Agda also happily accepts the following variant of *sort~*, which only provides the evidence, chaining the rewrite steps using transitivity, written as forward composition: $p \mathbin{⨾} q$.

```
sort~ : ∀ as → bag (sort as) ~ bag as
sort~ []         = ι
sort~ (a ∷ as)  =  insert~ a (sort as) ⨾ (ι ⊎ sort~ as)
```

Typical of after-the-fact verification, the proofs mirror the structure of the program: like *sort* resorts to a helper function, *sort~* relies on a lemma.

```
insert~ : ∀ a as → bag (insert a as) ~ ⟅ a ⟆ ⊎ bag as
insert~ a [] = ι
insert~ a (b ∷ bs) with a ⩽ ? ⩾′ b
... | LE     = ι
... | GE     = (ι ⊎ insert~ a bs) ⨾ swap
```

Sorting by insertion works by repeatedly swapping adjacent list elements. The proof reflects this property: the only non-trivial rearrangement is introduced by *swap*.

```
swap :    A ⊎ (B ⊎ C) ~ B ⊎ (A ⊎ C)
swap  =  σ α ⨾ (γ ⊎ ι) ⨾ α
```

Bag union is associative and commutative, evidenced by the combinators $\alpha : (A \uplus B) \uplus C \sim A \uplus (B \uplus C)$ and $\gamma : A \uplus B \sim B \uplus A$. In the first step above, we apply associativity from right to left (σ witnesses the symmetry of bag equivalence); then we swap the bags A and B; and, finally, we move the parentheses to the right.

$$A \uplus (B \uplus C) \sim (A \uplus B) \uplus C \sim (B \uplus A) \uplus C \sim B \uplus (A \uplus C)$$

The proof combinators are tangible. Applied to a random, four-element list *sort~* produces the following proof (lightly edited).

```
_ :   let as  =  4 ∷ 7 ∷ 1 ∷ 1 ∷ [] in bag (sort as) ~ bag as
_  =  (ι ⊎ swap) ⨾ swap ⨾ (ι ⊎ ((ι ⊎ swap) ⨾ swap))
```

The input list features four inversions. As to be expected, the number of swaps equals the inversion count.

As an intermediate summary, _~_ is an equivalence relation, it is reflexive, symmetric, and transitive. Bag union _⊎_ is associative and commutative with the empty bag ⟅⟆ as its neutral element. Figure 2 shows the API at a glance.

Bag membership. Turning briefly to the sorting property, the following inductively defined predicate captures that a list is ordered. (We assume that the type of elements is totally ordered.)

```
data _⩽-ordered_ (x : Elem) : List Elem → Set where
  []    : x ⩽-ordered []
  _∷_ : x ⩽ a → a ⩽-ordered as → x ⩽-ordered (a ∷ as)
```

Actually, the infix relation *x ⩽-ordered as* conjoins two properties: (1) *x* is a lower bound of the elements in *as*; and (2) the list *as* is non-descending. The second property holds by definition: the empty list is non-descending; a non-empty list is non-descending iff its head is a lower bound of the elements in the tail and the tail is itself non-descending. The first property, however, requires proof:

```
lower-bound : x ⩽-ordered as → x ⩽ ∗ bag as
lower-bound (pa ∷ pas) (head)   =  pa
lower-bound (pa ∷ pas) (tail p)  =  ⩽-transitive pa (lower-bound pas p)
```

The function makes use of a feature that we have not seen before: *bag membership*, written $a \in A$.

Membership is jolly useful when we want to quantify over elements of some *finite* collection. Assuming a function that "bagifies" a collection, *bag* : *Collection Elem* → *Bag Elem*, the statement ∀ *a* → *a* ∈ *bag as* → *P a* expresses that *P* holds for each element contained in the collection *as*. Likewise, ∃ *a* → *a* ∈ *bag as* × *P a*[1] captures the existence of an element in *as* that satisfies the property.

There are three principled ways to show membership: (1) *here* evidences that *a* is contained in the singleton bag ⟅ *a* ⟆; given either $p : x \in A$ or $q : x \in B$, (2) *inl p* and (3) *inr q* show that *x* is contained in $A \uplus B$. In general, the evidence for $a \in A$ points to an *occurrence* of *a* in *A*.

Membership is compatible with the equivalence relation: if *A* equivales *B*, then *x* is contained in *A* iff it is contained in *B*.

```
include   : A ~ B → (∀ {x} → x ∈ A → x ∈ B)
includeᴿ : A ~ B → (∀ {x} → x ∈ A ← x ∈ B)
```

Cancellation properties. Finite bags form a commutative monoid, the so-called *free commutative monoid*. This monoid is special in that it has two important cancellation properties, which allow us to cancel out common left or right terms.

```
cancel-left   : A ⊎ B₁ ~ A ⊎ B₂ → B₁ ~ B₂
cancel-right : A₁ ⊎ B ~ A₂ ⊎ B → A₁ ~ A₂
```

[1] Agda has no special support for existential quantification, so we actually have to write ∃ (λ *a* → *a* ∈ *bag as* × *P a*).

Operations

Constructors

Bags are parametrised by the element type:

$Bag : Set \to Set$

There are three principal constructors:

$\lbag\rbag : Bag\ X$
$\lbag\_\rbag : X \to Bag\ X$
$\_\uplus\_ : Bag\ X \to Bag\ X \to Bag\ X$

Functor and monad

The type constructor *Bag* is a monad:

$map : (X \to Y) \to (Bag\ X \to Bag\ Y)$
$\lbag\_\rbag : X \to Bag\ X$
$join : Bag\ (Bag\ X) \to Bag\ X$

Set-like operations

Standard "set-like" operations:

$\_\uplus\_ : Bag\ X \to Bag\ X \to Bag\ X$
$\_\cup\_ : Bag\ X \to Bag\ X \to Bag\ X \quad \star$
$\_\cap\_ : Bag\ X \to Bag\ X \to Bag\ X \quad \star$
$\_\setminus\_ : Bag\ X \to Bag\ X \to Bag\ X \quad \star$

Given evidence $r : x \in A$, "bag minus" $A - r$ removes this occurrence of x in A:

$\_-\_ : (A : Bag\ X) \to x \in A \to Bag\ X$

Queries

Cardinality and multiplicity:

$card : Bag\ X \to \mathbb{N}$
$\_\#\_ : X \to Bag\ X \to \mathbb{N} \quad \star$

Deciding membership, equivalence, and containment:

$\_\in?\_ : (x : X) \to (A : Bag\ X) \quad \star$
$\quad \to Decide\ (x \in A)$
$\_\sim?\_ : (A\ B : Bag\ X) \quad \star$
$\quad \to Decide\ (A \sim B)$
$\_\subseteq?\_ : (A\ B : Bag\ X) \quad \star$
$\quad \to Decide\ (A \subseteq B)$

Relations

Membership

Evidence for an occurrence of an element:

$here : x \in \lbag x \rbag$
$inl : x \in A \to x \in A \uplus B$
$inr : x \in B \to x \in A \uplus B$

Equivalence

Bag equivalence is, well, an equivalence:

$\iota : A \sim A$
$\sigma : A \sim B \to B \sim A$
$\_\mathbin{;}\_ : A \sim B \to B \sim C \to A \sim C$

Bag union is congruent:

$\_\uplus\_ : A_1 \sim A_2 \to B_1 \sim B_2$
$\quad \to A_1 \uplus B_1 \sim A_2 \uplus B_2$

Elements of equivalent bags can appear in any order:

$\lambda : \lbag\rbag \uplus A \sim A$
$\alpha : (A \uplus B) \uplus C \sim A \uplus (B \uplus C)$
$\gamma : A \uplus B \sim B \uplus A$

Evidence is abbreviated by greek letters so that proofs are short and sweet. Speaking names such as *∼-reflexive*, *⊎-congruent*, *∼-left-unit* are provided, as well.

Equivalence proofs

Wim Feijen's proof style allows you to make intermediate steps explicit, e.g.

```
ρ :   A ⊎ ⦃⦄ ∼ A
ρ =   proof
        A ⊎ ⦃⦄
      ∼⟨ γ ⟩
        ⦃⦄ ⊎ A
      ∼⟨ λ ⟩
        A
      ∎
```

Containment

The bag A is a subbag of B iff there exists a bag Δ such that $A \uplus \Delta \sim B$:

data $\_\subseteq\_ : Bag\ X \to Bag\ X \to Set$ **where**
$\quad \_such\text{-}that\_ : \forall\ \Delta \to A \uplus \Delta \sim B$
$\quad\quad \to A \subseteq B$

Containment is a partial order:

$\subseteq\text{-}reflexive : A \subseteq A$
$\sim\text{-}weaken : A \sim B \to A \subseteq B$
$\subseteq\text{-}transitive : A \subseteq B \to B \subseteq C \to A \subseteq C$
$\subseteq\text{-}antisymmetric : A \subseteq B \to B \subseteq A$
$\quad \to A \sim B$

Properties

Algebraic structures

- The type $Bag\ X$ is the free commutative monoid over X.
- It is conical and cancellative:

  $conical : A \uplus B \sim \lbag\rbag \to A \sim \lbag\rbag$
  $\sim\text{-}cancel : A \uplus B_1 \sim A \uplus B_2 \to B_1 \sim B_2$
  $\subseteq\text{-}cancel : A \uplus B_1 \subseteq A \uplus B_2 \to B_1 \subseteq B_2$

- $(Bag\ X, \_\cup\_, \_\cap\_)$ is a distributive lattice: $\_\cup\_$ and $\_\cap\_$ are associative, commutative, and idempotent; they satisfy absorption and distributive laws:

  $A \cup (A \cap B) \sim A$
  $A \cap (A \cup B) \sim A$
  $A \cup (B \cap C) \sim (A \cup B) \cap (A \cup C)$
  $A \cap (B \cup C) \sim (A \cap B) \cup (A \cap C)$

- $\lbag\rbag$ is the least element:

  $\lbag\rbag \cup A \sim A$
  $\lbag\rbag \cap A \sim \lbag\rbag$

- $\_\uplus\_$ is associative and commutative with the empty bag as its neutral element; $\_\uplus\_$ distributes over $\_\cup\_$ and $\_\cap\_$:

  $A \uplus (B \cap C) \sim (A \uplus B) \cap (A \uplus C)$
  $A \uplus (B \cup C) \sim (A \uplus B) \cup (A \uplus C)$

- Absorption laws and counting law:

  $A \cap (A \uplus B) \sim A$
  $A \cup (A \uplus B) \sim A \uplus B$
  $A \uplus B \quad \sim (A \cup B) \uplus (A \cap B)$

Bag difference

Bag sum $\_\uplus P$ has a left adjoint, bag difference $\_\setminus P$, but not a right adjoint.

$(A \setminus P \subseteq B) \leftrightarrow (A \subseteq B \uplus P)$
$(A \uplus P \subseteq B) \to (A \subseteq B \setminus P)$
$(A \uplus P \subseteq B) \leftarrow (A \subseteq B \setminus P) \leftarrow P \subseteq B$

$to\text{-}rear : (r : x \in A)$
$\quad \to A \sim (A - r) \uplus \lbag x \rbag$

Interrelations

Connecting lemma:

$A \cap B \sim A \ \leftrightarrow\ A \subseteq B \ \leftrightarrow\ A \cup B \sim B$

Membership is related to equivalence and containment:

$\sim\text{-}include : A \sim B \to x \in A \to x \in B$
$\subseteq\text{-}include : A \subseteq B \to x \in A \to x \in B$
$x \in A \ \leftrightarrow\ \lbag x \rbag \subseteq A$

Arithmetic pecking order:

$A \cap B \subseteq A \cup B \subseteq A \uplus B$

Patterns of definition

Functions from bags

Functions from bags can be defined using pattern matching:

$f : Bag\ X \to Y$
$f\ \lbag\rbag = \ldots$
$f\ \lbag x \rbag = \ldots$
$f\ (A \uplus B) = \ldots$

CM-homomorphisms are captured by

$fold : \{\!\{ eq : Eq\ Y \}\!\}$
$\quad \{\!\{ M : Commutative\text{-}Monoid\ Y \}\!\}$
$\quad \to (X \to Y) \to Bag\ X \to Y$

Freeness entails that $fold\ f$ is the *unique* CM-homomorphism from the bag monoid to M such that $fold\ f\ \lbag x \rbag \approx f\ x$.

Views

The "cons view" allows us to distinguish between empty and non-empty bags:

data $View\ (A : Bag\ X) : Set$ **where**
$\quad Empty : A \sim \lbag\rbag \to View\ A$
$\quad Laden : (a : X)$
$\quad\quad \to (A' : Bag\ X)$
$\quad\quad \to A \sim \lbag a \rbag \uplus A'$
$\quad\quad \to View\ A$

$select : (A : Bag\ X) \to View\ A$

If the bag is non-empty, it is split into an element and the residual bag.
The "list view" iterates the cons view:

data $List\text{-}View\ (A : Bag\ X) : Set$ **where**
$\quad [\,] : \{\phi : A \sim \lbag\rbag\} \to List\text{-}View\ A$
$\quad \_::\_ : (a : X)$
$\quad\quad \to \{A' : Bag\ X\}$
$\quad\quad \to \{\phi : A \sim \lbag a \rbag \uplus A'\}$
$\quad\quad \to List\text{-}View\ A' \to List\text{-}View\ A$

$list\text{-}view : (A : Bag\ X)$
$\quad \to List\text{-}View\ A$

Further properties

Multiplicities

$x \in A \ \leftrightarrow\ x \# A \geqslant 1$
$A \sim B \ \leftrightarrow\ (\forall\ x \to x \# A \equiv x \# B)$
$A \subseteq B \ \leftrightarrow\ (\forall\ x \to x \# A \leqslant x \# B)$
$x \# (A \uplus B) \equiv (x \# A) + (x \# B)$
$x \# (A \setminus B) \equiv (x \# A) \dot{-} (x \# B)$
$x \# (A \cap B) \equiv (x \# A) \downarrow (x \# B)$
$x \# (A \cup B) \equiv (x \# A) \uparrow (x \# B)$

Figure 2: The "Bag API Cheat Sheet" (operations marked with a "★" require decidable equality)

To illustrate the use of cancellation, let us show that two ordered permutations are equal: if x $\leqslant$*-ordered* as and x $\leqslant$*-ordered* bs, then

$$\mathit{bag}\ \mathit{as} \sim \mathit{bag}\ \mathit{bs} \leftrightarrow \mathit{as} \equiv \mathit{bs}$$

The right-to-left direction holds trivially as propositional equality $\_{\equiv}\_$ is the least reflexive relation. Figure 3 lists the proof for the other direction. We sketch the argument for the interesting case that both lists are non-empty, that is, $\mathit{as} \coloneqq (a :: \mathit{as})$ and $\mathit{bs} \coloneqq (b :: \mathit{bs})$. Since we know that b itself is a lower bound of $b :: \mathit{bs}$,

$$\begin{aligned}
&\mathit{pbs} &&: b \leqslant\text{-}\mathit{ordered}\ \mathit{bs}\\
&(\leqslant\text{-}\mathit{reflexive} :: \mathit{pbs}) &&: b \leqslant\text{-}\mathit{ordered}\ (b :: \mathit{bs})
\end{aligned}$$

and, furthermore, that a is contained in $b :: \mathit{bs}$,

$$\begin{aligned}
&\phi &&: (a :: \mathit{as}) \sim (b :: \mathit{bs})\\
&(\mathit{inl}\ \mathit{here}) &&: a \in (a :: \mathit{as})\\
&(\mathit{include}\ \phi\ (\mathit{inl}\ \mathit{here})) &&: a \in (b :: \mathit{bs})
\end{aligned}$$

we may conclude using *lower-bound* that $b \leqslant a$. A symmetric argument gives $a \leqslant b$. Since total orders are antisymmetric, the two heads are propositionally equal: $a \equiv b$. Cancelling the heads, the equality of the tails is then established recursively.

Specification of abstract datatypes. Bags also play a pivotal role in the specification of abstract datatypes. Take for example priority queues, which support efficient access to the least element of a collection. Priority queues are conceptually bags as the collection may contain repeated elements. Assuming that queues are indexed by their size, $\mathbb{Q} : \mathbb{N} \to \mathit{Set}$, a fairly complete specification of the requirements is given by:

$$\begin{array}{lll}
 & & \mathit{bag}\ \mathit{empty} \sim \Lbag\Rbag\\
\forall\ a & \to & \mathit{bag}\ (\mathit{singleton}\ a) \sim \Lbag\ a\ \Rbag\\
\forall\ (P : \mathbb{Q}\ m)\ (Q : \mathbb{Q}\ n) & \to & \mathit{bag}\ (P\ \mathit{union}\ Q) \sim \mathit{bag}\ P \uplus \mathit{bag}\ Q\\
\forall\ (Q : \mathbb{Q}\ 0) & \to & \Lbag\Rbag \sim \mathit{bag}\ Q\\
\forall\ (Q : \mathbb{Q}\ (1+n)) & \to & \Lbag\ \mathit{min}\ Q\ \Rbag \uplus \mathit{bag}\ (\mathit{delete\text{-}min}\ Q) \sim \mathit{bag}\ Q\\
\forall\ (Q : \mathbb{Q}\ (1+n)) & \to & (\forall\ \{a\} \to a \in \mathit{bag}\ Q \to \mathit{min}\ Q \leqslant a)
\end{array}$$

The first three equivalences express that the queue constructors are concrete incarnations of the corresponding abstract bag constructors. Non-empty queues of type $\mathbb{Q}\ (1+n)$ support computing the minimal element, $\mathit{min} : \mathbb{Q}\ (1+n) \to \mathit{Elem}$, and removing it, $\mathit{delete\text{-}min} : \mathbb{Q}\ (1+n) \to \mathbb{Q}\ n$. The last two requirements ensure that $\mathit{min}\ Q$ is the least element of Q: it is contained in Q and a lower bound for the elements in Q. Overall, the axioms guarantee that each extracted element was added beforehand.

3 REPRESENTATION OF BAGS

As for the implementation of bags, we learned the following lesson: *don't try to be clever.* The simplest, most straightforward implementation turned out to be the most practical in terms of proof effort and convenience of use. This section provides the nitty-gritty details, including discussions of alternative approaches.

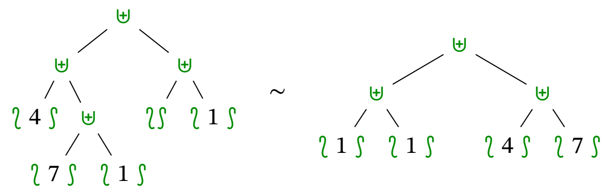
Recall that there are three principled ways for forming bags: singleton bags $\Lbag\ x\ \Rbag$, the empty bag $\Lbag\Rbag$, and bag union $A \uplus B$. We turn these operations into data constructors of the *Bag* datatype.

$$\begin{aligned}
&\mathbf{data}\ \mathit{Bag}\ (X : \mathit{Set}) : \mathit{Set}\ \mathbf{where}\\
&\quad \Lbag\_\Rbag : X \to \mathit{Bag}\ X\\
&\quad \Lbag\Rbag \quad\ : \mathit{Bag}\ X\\
&\quad \_\uplus\_ \ \ : \mathit{Bag}\ X \to \mathit{Bag}\ X \to \mathit{Bag}\ X
\end{aligned}$$

Loosely speaking, the operations do nothing, they merely create binary trees, variably known as *join lists* or *leaf trees*. An alternative view is to interpret the elements of *Bag X* as bag expressions or syntax trees — so the datatype definition captures the *syntax* of bags. For example, the bags

$$\begin{aligned}
&\mathit{Eau\text{-}de\text{-}Cologne}\ \mathit{East\text{-}Berlin} : \mathit{Bag}\ \mathbb{N}\\
&\mathit{Eau\text{-}de\text{-}Cologne} = (\Lbag\,4\,\Rbag \uplus (\Lbag\,7\,\Rbag \uplus \Lbag\,1\,\Rbag)) \uplus (\Lbag\Rbag \uplus \Lbag\,1\,\Rbag)\\
&\mathit{East\text{-}Berlin} \qquad\ = (\Lbag\,1\,\Rbag \uplus \Lbag\,1\,\Rbag) \uplus (\Lbag\,4\,\Rbag \uplus \Lbag\,7\,\Rbag)
\end{aligned}$$

correspond to the trees depicted below.



Both binary trees represent the same "abstract" bag: *Eau-de-Cologne* equivales *East-Berlin* (more about equivalence shortly). In general, a finite bag has many concrete tree representations, in fact, infinitely many due to the availability of empty leaves.

The type of finite bags is a functor

$$\begin{aligned}
&\mathit{map} : (X \to Y) \to (\mathit{Bag}\ X \to \mathit{Bag}\ Y)\\
&\mathit{map}\ f\ \Lbag\ x\ \Rbag &&= \Lbag\ f\ x\ \Rbag\\
&\mathit{map}\ f\ \Lbag\Rbag &&= \Lbag\Rbag\\
&\mathit{map}\ f\ (A \uplus B) &&= \mathit{map}\ f\ A \uplus \mathit{map}\ f\ B
\end{aligned}$$

and a monad: $\Lbag\_\Rbag$ is its unit, multiplication is defined:

$$\begin{aligned}
&\mathit{join} : \mathit{Bag}\ (\mathit{Bag}\ X) \to \mathit{Bag}\ X\\
&\mathit{join}\ \Lbag\ A\ \Rbag &&= A\\
&\mathit{join}\ \Lbag\Rbag &&= \Lbag\Rbag\\
&\mathit{join}\ (A \uplus B) &&= \mathit{join}\ A \uplus \mathit{join}\ B
\end{aligned}$$

Remark 1. *A common alternative [10, 13] is to represent bags as functions into the natural numbers, Bag X = X → ℕ, mapping each* potential *element to its* multiplicity*. The empty bag, various "set-like" operations and relations enjoy elegant definitions, which is perhaps why the approach is tempting (*$\_\dot{-}\_$ *is subtraction of natural numbers aka monus;* $\_{\uparrow}\_$ *and* $\_{\downarrow}\_$ *are maximum and minimum, respectively).*

$$\begin{array}{rlrl}
\Lbag\Rbag = & \lambda\ x \to 0 & A \uplus B = & \lambda\ x \to A\ x + B\ x\\
A \subseteq B = & \forall\ x \to A\ x \leqslant B\ x & A \setminus B = & \lambda\ x \to A\ x \dot{-} B\ x\\
A \sim B = & \forall\ x \to A\ x \equiv B\ x & A \cup B = & \lambda\ x \to A\ x \uparrow B\ x\\
 & & A \cap B = & \lambda\ x \to A\ x \downarrow B\ x
\end{array}$$

However, other operations are less convenient or even impossible to define. For example, for singleton bags we need to assume that equality on X is decidable (see also Section 7). Bags as functions into the naturals comprise both finite and infinite bags, so this is actually a different type! In particular, it fails to be a monad as, for example, the nested bag $\Lbag\ \Lbag\Rbag, \Lbag\,1\,\Rbag, \Lbag\,1, 2\,\Rbag, \Lbag\,1, 2, 3\,\Rbag, \ldots\ \Rbag$ *cannot be flattened. Worse, it is not possible to define functions out of bags, such as cardinality (a bag is like a black hole).*

```
unique : x ≼-ordered as → x ≼-ordered bs → bag as ∼ bag bs → as ≡ bs
unique []          []          ϕ  =  reflexive
unique []          (pb :: pbs) ϕ  =  impossible ϕ
unique (pa :: pas) []          ϕ  =  impossible (σ ϕ)
unique (pa :: pas) (pb :: pbs) ϕ
  with ≼-antisymmetric (lower-bound (≼-reflexive :: pbs) (include  ϕ (inl here)))
                       (lower-bound (≼-reflexive :: pas) (includeᴿ ϕ (inl here)))
... | reflexive  =  congruent₂ _::_ reflexive (unique pas pbs (cancel-left ϕ))
```

Figure 3: Use of cancellation: two ordered permutations are equal.

Remark 2. *The support of a bag is the* set *of elements with positive multiplicity.* Finite bags *can be modelled as functions into the naturals with finite support. While this definition works well in theory it is less useful in practice as it forces us to implement finite sets first.*

Alternatively, we could model finite bags as finite maps *into the naturals, building on an arbitrary implementation of finite maps, such as search trees or tries. However, then we need to make even stronger assumptions, typically, that the element type is totally ordered. Moreover: We intend, perhaps, to use bags to establish the correctness of search trees — we have to be careful not to create a vicious circle.*

4 RELATIONS ON BAGS

Membership. For the purposes of presentation, let us assume that equality on the element type X is given by propositional equality. (This assumptions is *not* acceptable in the production code as it would bar us from using bags of bags, see also Section 10.)

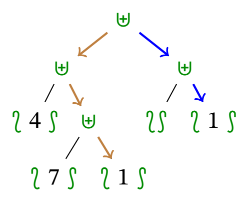
Bag membership is an inductively defined relation.

```
data _∈_ (x : X) : Bag X → Set where
  here : x ∈ ≀ x ≀
  inl  : x ∈ A → x ∈ A ⊎ B
  inr  : x ∈ B → x ∈ A ⊎ B
```

An element of $x \in A$ can be seen as a *path* in the tree A, a path from the root to some x-labelled leaf. For example, the number 1 occurs twice in *Eau-de-Cologne*, so there are two different proofs of membership,

```
one one′ : 1 ∈ Eau-de-Cologne
one   =  inl (inr (inr here))
one′  =  inr (inr here)
```

illustrated below, where the first path is highlighted in brown and the second in blue.



In general, the number of paths of type $x \in A$ equals the multiplicity of the element x in A.

Equivalence. Since bags are trees in disguise, we cannot use propositional equality as the underlying equivalence as it is too discriminating. Rather, we define a tailor-made relation. Similar to the definition of *Bag*, we turn the defining properties of the relation into constructors of a datatype. In other words, we define an inference or deduction system, whose elements are proof trees.

```
data _∼_ : Bag X → Bag X → Set where
  ι    : A ∼ A
  σ    : A ∼ B → B ∼ A
  _;_  : A ∼ B → B ∼ C → A ∼ C
  _⊎_  : A₁ ∼ A₂ → B₁ ∼ B₂ → A₁ ⊎ B₁ ∼ A₂ ⊎ B₂
  ƛ    : ≀≀ ⊎ A ∼ A
  α    : (A ⊎ B) ⊎ C ∼ A ⊎ (B ⊎ C)
  γ    : A ⊎ B ∼ B ⊎ A
```

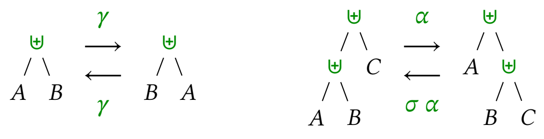
The axioms of the inference system can be divided into three groups: (1) the first three axioms capture that the relation is an equivalence; (2) the fourth axiom specifies that it is a congruence: bag union applied to equivalent arguments yields equivalent results; and, finally, (3) the last three axioms determine that bag union is associative and commutative with the empty bag as its neutral element.

Observe that we do not include right unitality as an axiom as it can be easily inferred:

```
ρ :   A ⊎ ≀≀ ∼ A
ρ  =  γ ; ƛ
```

As an aside, the proof format used in Section 2 is taken from the Agda standard library [14] — the notation is completely standard and can be defined for any transitive relation.

A bag is a tree, evidence for membership is a path. In this spirit, evidence for the equivalence of two bags is a *tree transformer* that transmogrifies the first into the second tree. For example, γ swaps two subtrees (incidentally, a typical operation used in the implementation of priority queues), whereas α performs a right rotation (incidentally, a typical operation used in the implementation of search trees or sequences).



Continuing our running example, the proof

```
go-east :  Eau-de-Cologne ∼ East-Berlin
go-east  =  (σ α ⊎ ƛ) ; α ; γ
```

shows how to transmogrify *Eau-de-Cologne* to *East-Berlin* — Figure 4 details each step of the transformation.

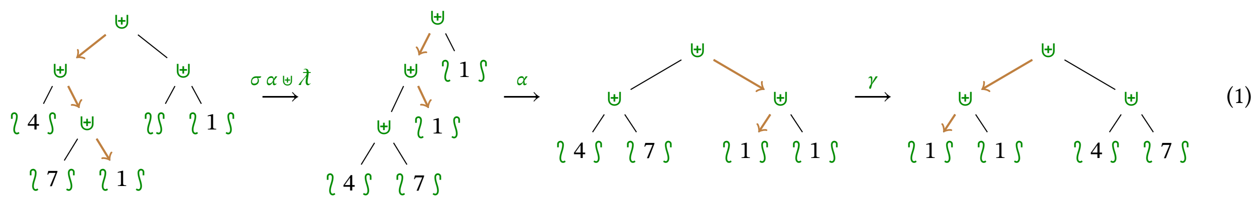


Figure 4: A proof that *Eau-de-Cologne* equivales *East-Berlin* — the transformation sends the leftmost occurrence of 1 in the first tree to the leftmost occurrence of 1 in the second tree.

Remark 3. *The equivalence relation can be defined in a variety of different ways. An alternative approach builds on the fact that two bags are equivalent iff the multiplicities of each element are equal (see Remark 1). Since furthermore the multiplicity of an element x in A is given by the number of paths of type $x \in A$, we could define:*

$$A \approx B \;=\; \forall\,\{x\} \to (x \in A) \cong (x \in B) \qquad (2)$$

This approach is known as "proof-relevant membership" [8], as the proofs of $x \in A$ and $x \in B$ have to be in one-to-one correspondence. This approach is, however, cumbersome in practise as it requires us to define a mapping from $x \in A$ to $x \in B$ plus proofs that the mapping is injective and surjective or, alternatively, to exhibit a forward and a backward mapping plus proofs that they are mutually inverse. In what follows we show that $A \sim B$ implies $A \approx B$. The reader is invited to establish the other direction, which is a lot harder.

A tree transformation induces a transformation of paths, actually two transformations, one in the forward direction and another one in the backward direction.

```
include   : A ∼ B → (∀ {x} → x ∈ A → x ∈ B)
includeᴿ  : A ∼ B → (∀ {x} → x ∈ A ← x ∈ B)
```

We only show the proof of the forward direction. Its counterpart is defined completely analogously. In fact, the functions are mutually recursive: in the σ-case, one calls the other (a typical setup for functions that operate on proof trees).

```
include ι        p              = p
include (σ ϕ)    p              = includeᴿ ϕ p
include (ϕ ⨾ ψ)  p              = include ψ (include ϕ p)
include (ϕ ⊎ ψ)  (inl p)        = inl (include ϕ p)
include (ϕ ⊎ ψ)  (inr q)        = inr (include ψ q)
include ƛ        (inr p)        = p
include α        (inl (inl p))  = inl p
include α        (inl (inr q))  = inr (inl q)
include α        (inr r)        = inr (inr r)
include γ        (inl p)        = inr p
include γ        (inr q)        = inl q
```

Consider the equations dealing with associativity "α": the law involves three variables, the corresponding trees consist of three named subtrees, hence we have three equations dealing with three different paths. Similar remarks apply to the other equations.

Two routine proofs then show that the transformations are mutually inverse.

```
includeᴿ ϕ (include  ϕ p) ≡ p
include  ϕ (includeᴿ ϕ q) ≡ q
```

To illustrate, the leftmost 1 in *Eau-de-Cologne* is mapped to the leftmost 1 in *East-Berlin*, see also Figure 4.

```
_ :  include go-east one ≡ inl (inl here)
_ = reflexive
```

Containment. Each monoid $(M; \varepsilon; \bullet)$ comes equipped with a pre-ordering, the so-called *algebraic ordering*: a is at most b if and only if there is a Δ such that $a \bullet \Delta = b$. If the monoid is additionally cancellative and conical (see Sections 5 and 6), then the preorder is antisymmetric. In other words, the algebraic ordering is actually a partial order. We use the general construction to define the subbag relation.

```
data _⊆_ : Bag X → Bag X → Set where
  _such-that_ : ∀ Δ → A ⊎ Δ ∼ B → A ⊆ B
_⊇_ : Bag X → Bag X → Set
A ⊇ B  =  B ⊆ A
```

Reflexivity and transitivity enjoy straightforward (generic) proofs.

```
⊆-reflexive :  A ⊆ A
⊆-reflexive  =  ⟅⟆ such-that γ ⨾ ƛ
⊆-transitive : A ⊆ B → B ⊆ C → A ⊆ C
⊆-transitive (Δ₁ such-that ϕ₁) (Δ₂ such-that ϕ₂)  =
  Δ₁ ⊎ Δ₂ such-that σ α ⨾ (ϕ₁ ⊎ ι) ⨾ ϕ₂
```

We postpone the proof of antisymmetry until Section 6.

Summary. The set of all finite bags with elements drawn from X forms a commutative monoid. In fact, the structure is a *free* commutative monoid on X (more about this in Section 5). The implementation basically amounts to the free structure: expressions built from the empty bag ⟅⟆, singleton bags ⟅ x ⟆, and bag union $A \uplus B$, quotiented by the laws of commutative monoids.

Quite attractively, the very same approach can be used to implement lists, the free monoid, and finite sets, the free idempotent commutative monoid aka the free bounded semilattice. For lists, we simply drop commutativity, $A +\!\!+ B \sim B +\!\!+ A$. For sets, we simply add idempotency, $A \cup A \sim A$.

5 OPERATIONS ON BAGS

Operations on bags are defined by induction over the structure of the bag datatype. Consider, as an example, the operation that removes a single *occurrence* of an element in a given bag.

```
_−_ : (A : Bag X) → (r : x ∈ A) → Bag X
⟅ x ⟆     − here  =  ⟅⟆
(A ⊎ B) − inl p  =  (A − p) ⊎ B
(A ⊎ B) − inr q  =  A ⊎ (B − q)
```

Observe the difference to general bag difference (pun intended). Since we definitely know that the element x occurs in A, adding x again yields the original bag A.

```
to-rear : (r : x ∈ A) → A ∼ (A − r) ⊎ ⟅ x ⟆
to-rear (here)   =  σ ƛ
to-rear (inl p)  =  (to-rear p ⊎ ι) ⨾ α ⨾ (ι ⊎ γ) ⨾ σ α
to-rear (inr q)  =  (ι ⊎ to-rear q) ⨾ σ α
```

We use removal to implement general bag difference, postponing the definition until Section 7.

Homomorphisms. Monoid homomorphisms are particularly easy to define. In Section 2 we have introduced a transformation that turns a list into a bag. Here is its inverse:

```
list : Bag X → List X
list ⟅ x ⟆      =  [ x ]
list ⟅⟆         =  []
list (A ⊎ B)  =  list A ++ list B
```

In general, a homomorphism is uniquely defined by its action on singleton bags; the second and third equation are forced as a homomorphism preserves the structure of a monoid: bag union is mapped to the operation of the target monoid and the empty bag to its unit.

As a further example, *card* determines the cardinality or size of a finite bag, the sum of all multiplicities.

```
card : Bag X → ℕ
card ⟅ x ⟆      =  1
card ⟅⟆         =  0
card (A ⊎ B)  =  card A + card B
```

Cardinality is even a homomorphism between *commutative* monoids, a *CM-homomorphism.* This entails, however, a proof obligation. We have to show that *card* does not depend on the representation of its argument: equivalent bags possess the same cardinality.

```
card∼ : A₁ ∼ A₂ → card A₁ ≡ card A₂
card∼ (ι)                =  reflexive
card∼ (σ ϕ)              =  symmetric (card∼ ϕ)
card∼ (ϕ ⨾ ψ)            =  transitive (card∼ ϕ) (card∼ ψ)
card∼ (ϕ ⊎ ψ)            =  congruent₂ _+_ (card∼ ϕ) (card∼ ψ)
card∼ (ƛ)                =  +-left-unit _
card∼ (α {A = A})  =  +-associative (card A)
card∼ (γ {A = A})  =  +-commutative (card A) _
```

Like the axioms of _∼_, the proof obligations can be divided into three groups: (1) the first three equations use that propositional equality is an equivalence; (2) the fourth equation uses that addition is compatible with this relation; and, finally, (3) the last three equations confirm that addition is associative and commutative with 0 as its neutral element. Everything nicely falls into place.

By contrast, *list* is not a CM-homomorphism as list concatenation is not commutative. In other words, *list* is sensitive to the representation of the to-be-listed bag. There is nothing wrong with representation dependence per se; the function *list* is just less widely applicable compared to a true CM-homomorphism.

Cardinality is useful for conducting arguments over the size of a bag. As an example, let us show that the bag monoid is conical. In general, a monoid $(M; \varepsilon; \bullet)$ is *conical* iff $a \cdot b = \varepsilon$ implies $a = \varepsilon$ and $b = \varepsilon$ for all a and b. In other words, non-ε elements have no inverse. We first establish a useful lemma: a bag with cardinality zero equivales the empty bag.

```
card-A≡0→A∼⟅⟆ : ∀ (A : Bag X) → card A ≡ 0 → A ∼ ⟅⟆
card-A≡0→A∼⟅⟆ ⟅⟆         ϕ  =  ι
card-A≡0→A∼⟅⟆ (A ⊎ B) ϕ  =
  (  card-A≡0→A∼⟅⟆ A (m+n≡0→m≡0 ϕ)
  ⊎ card-A≡0→A∼⟅⟆ B (m+n≡0→n≡0  ϕ)) ⨾ ƛ
```

The proofs basically utilise that the monoid of natural numbers with addition is itself conical.

```
conical : A ⊎ B ∼ ⟅⟆ → A ∼ ⟅⟆
conical ϕ  =  card-A≡0→A∼⟅⟆ _ (m+n≡0→m≡0 (card∼ ϕ))
```

We have noted that the bag monoid (*Bag X*; _∼_; ⟅⟆; _⊎_) is the free commutative monoid (CM) generated by X. Categorically speaking, the free structure is part of an adjunction between the category of CMs and CM-homomorphisms and the category of setoids and extensional functions. The adjunction entails that CM-homomorphisms *from* the free CM over X to some other CM M are in one-to-one correspondence to functions from X *to* the carrier of M. We materialise one direction of this correspondence as a general recursion scheme, called, well, *fold*. For this purpose, the operations and properties of CMs are combined in a record type.

```
record Commutative-Monoid (X : Set) ⦃ eq : Eq X ⦄ : Set where
field
  ε            : X
  _·_          : X → X → X
  ·-congruent  : ∀ {x x′ y y′} → x ≈ x′ → y ≈ y′ → x · y ≈ x′ · y′
  left-unit    : ∀ {x}       → ε · x ≈ x
  commutative : ∀ {x y}     → x · y ≈ y · x
  associative  : ∀ {x y z} → (x · y) · z ≈ x · (y · z)
right-unit     : ∀ {x}       → x · ε ≈ x
right-unit  =  transitive commutative left-unit
open Commutative-Monoid ⦃...⦄ public
```

Observe that the record type is parametrised by the carrier *and* an equivalence relation (think Haskell's *Eq* class extended by properties) so that we can turn bags into suitable instances.

```
instance
  Free≈ : Eq (Bag X)
  Free≈  =  record {_≈_  =  _∼_; reflexive  =  ι;
                         transitive  =  _⨾_; symmetric  =  σ}
  Free : Commutative-Monoid (Bag X)
  Free  =  record {_·_  =  _⊎_; ·-congruent  =  _⊎_; ε  =  ∅;
                       associative  =  α; left-unit  =  ƛ; commutative  =  γ}
```

The fold operator turns a function, the action on singletons, into a CM-homomorphism.

$$\begin{aligned}
&\mathit{fold} : \{\!\{\, \mathit{eq} : \mathit{Eq}\ Y \,\}\!\}\ \{\!\{\, M : \mathit{Commutative\text{-}Monoid}\ Y \,\}\!\} \to\\
&\quad (X \to Y) \to \mathit{Bag}\ X \to Y\\
&\mathit{fold}\ f\ \Lbag\, x \,\Rbag \quad = f\ x\\
&\mathit{fold}\ f\ \Lbag\Rbag \quad\;\; = \varepsilon\\
&\mathit{fold}\ f\ (A \uplus B) = \mathit{fold}\ f\ A \cdot \mathit{fold}\ f\ B
\end{aligned}$$

It is plain to see that *fold f* is the *unique* CM-homomorphism such that $\mathit{fold}\ f\ \Lbag\, x \,\Rbag \approx f\ x$. Furthermore, *fold f* is independent of the representation of bags:

$$\begin{aligned}
&\mathit{fold}{\sim} : \{\!\{\, \mathit{eq} : \mathit{Eq}\ Y \,\}\!\}\ \{\!\{\, M : \mathit{Commutative\text{-}Monoid}\ Y \,\}\!\} \to\\
&\quad (f : X \to Y) \to A \sim B \to \mathit{fold}\ f\ A \approx \mathit{fold}\ f\ B\\
&\mathit{fold}{\sim}\ f\ \iota &&= \mathit{reflexive}\\
&\mathit{fold}{\sim}\ f\ (\sigma\ \phi) &&= \mathit{symmetric}\ (\mathit{fold}{\sim}\ f\ \phi)\\
&\mathit{fold}{\sim}\ f\ (\phi\ ;\ \psi) &&= \mathit{transitive}\ (\mathit{fold}{\sim}\ f\ \phi)\ (\mathit{fold}{\sim}\ f\ \psi)\\
&\mathit{fold}{\sim}\ f\ (\phi \uplus \psi) &&= \cdot\text{-}\mathit{congruent}\ (\mathit{fold}{\sim}\ f\ \phi)\ (\mathit{fold}{\sim}\ f\ \psi)\\
&\mathit{fold}{\sim}\ f\ \text{ƛ} &&= \mathit{left\text{-}unit}\\
&\mathit{fold}{\sim}\ f\ \alpha &&= \mathit{associative}\\
&\mathit{fold}{\sim}\ f\ \gamma &&= \mathit{commutative}
\end{aligned}$$

Now, assuming a suitable instance for natural numbers $\mathbb{N}$ with addition, cardinality is simply given by $\mathit{fold}\ (\lambda\ x \to 1)$.

6 CANCELLATION PROPERTIES

The proof of the cancellation properties is the litmus test for any implementation of bags. (Originally, we used an implementation based on proof-relevant membership, see Remark 3. Proving cancellation for this representation proved too cumbersome. As a consequence, we discarded the approach and opted for the representation described in this paper.)

The proof of left cancellation proceeds by induction over the structure of the to-be-cancelled term.

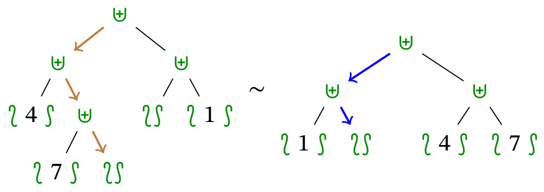
$$\begin{aligned}
&\mathit{cancel\text{-}left} : \forall\ A \to A \uplus B_1 \sim A \uplus B_2 \to B_1 \sim B_2\\
&\mathit{cancel\text{-}left}\ \Lbag\, x \,\Rbag \quad\ \phi = \sigma\ \text{ƛ}\ ;\ \mathit{minus}{\sim}\ (\mathit{inl}\ \mathit{here})\ (\mathit{inl}\ \mathit{here})\ \phi\ ;\ \text{ƛ}\\
&\mathit{cancel\text{-}left}\ \Lbag\Rbag \qquad \phi = \sigma\ \text{ƛ}\ ;\ \phi\ ;\ \text{ƛ}\\
&\mathit{cancel\text{-}left}\ (A_1 \uplus A_2)\ \phi = \mathit{cancel\text{-}left}\ A_2\ (\mathit{cancel\text{-}left}\ A_1\ (\sigma\ \alpha\ ;\ \phi\ ;\ \alpha))
\end{aligned}$$

If the term is empty, there is little to do. If the term is a bag union, we recursively cancel the terms, one after the other. Perhaps surprisingly, the final case, $\Lbag\, x \,\Rbag \uplus B_1 \sim \Lbag\, x \,\Rbag \uplus B_2$, requires work. It is less straightforward than it seems, because the equivalence not necessarily relates the visible occurrence of x on the left (*inl here*) to the visible occurrence on the right (*inl here*). In general, B_1 and B_2 may contain further occurrences of x. This motivates the following generalisation of the singleton case.

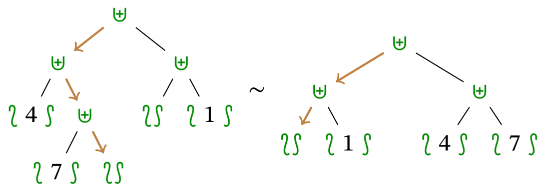
$$\begin{aligned}
&\mathit{minus}{\sim} : (p : x \in A) \to (q : x \in B) \to A \sim B \to A - p \sim B - q\\
&\mathit{minus}{\sim}\ p\ q\ A{\sim}B\ = \mathit{copy}\ p\ A{\sim}B\ ;\ \mathit{trade}\ (\mathit{include}\ A{\sim}B\ p)\ q
\end{aligned}$$

In words: given two equivalent bags, A and B, an occurrence of x in A, an unrelated occurrence of x in B, removing these occurrences preserves equivalence. To prove the generalised law, we proceed in two steps. The lemma *copy* establishes the special case that the path is the „same“, but the trees are different. The lemma *trade* deals with the symmetric situation that the paths are different, but the tree is the same.

A few pictures probably would not go amiss. Continuing our running example, *Eau-de-Cologne* ∼ *East-Berlin* (1), we remove an occurrence of the number 1 on both sides, aiming to show:



Recall that the given tree transformation (1), see Figure 4, induces a path transformation, sending the leftmost occurrence of 1 in the first tree to the leftmost occurrence of 1 in the other tree. So in the first step, we establish



The proof of this equivalence is actually a no-brainer: the equivalence is identical to the original one (1), except for the types. (Do you see why?) A no-brainer, but laborious as we have to replay the case analysis of *include*. Consequently, we define two mutually recursive functions:

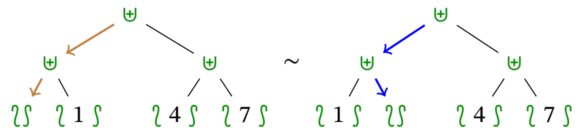
$$\begin{aligned}
&\mathit{copy}\ \ : (p : x \in A) \to (\phi : A \sim B) \to A - p \sim B - \mathit{include}\ \phi\ p\\
&\mathit{copy}^R : (q : x \in B) \to (\phi : A \sim B) \to A - \mathit{include}^R\ \phi\ q \sim B - q
\end{aligned}$$

Both definitions are rather boring so we only show the first one.

$$\begin{aligned}
&\mathit{copy}\ p && \iota && = \iota\\
&\mathit{copy}\ p && (\sigma\ \phi) && = \sigma\ (\mathit{copy}^R\ p\ \phi)\\
&\mathit{copy}\ p && (\phi\ ;\ \psi) && = \mathit{copy}\ p\ \phi\ ;\ \mathit{copy}\ (\mathit{include}\ \phi\ p)\ \psi\\
&\mathit{copy}\ (\mathit{inl}\ p) && (\phi \uplus \psi) && = \mathit{copy}\ p\ \phi \uplus \psi\\
&\mathit{copy}\ (\mathit{inr}\ q) && (\phi \uplus \psi) && = \phi \uplus \mathit{copy}\ q\ \psi\\
&\mathit{copy}\ (\mathit{inr}\ p) && \text{ƛ} && = \text{ƛ}\\
&\mathit{copy}\ (\mathit{inl}\ (\mathit{inl}\ p)) && \alpha && = \alpha\\
&\mathit{copy}\ (\mathit{inl}\ (\mathit{inr}\ q)) && \alpha && = \alpha\\
&\mathit{copy}\ (\mathit{inr}\ r) && \alpha && = \alpha\\
&\mathit{copy}\ (\mathit{inl}\ p) && \gamma && = \gamma\\
&\mathit{copy}\ (\mathit{inr}\ q) && \gamma && = \gamma
\end{aligned}$$

Clearly, *copy* is morally the identity.

As for the second step, it remains to show:



The proof proceeds by simultaneous induction over both paths.

$$\begin{aligned}
&\mathit{paws} : (A \uplus B) \uplus C \sim A \uplus (C \uplus B)\\
&\mathit{paws}\ = \alpha\ ;\ \iota \uplus \gamma\\
&\mathit{trade} : (p : x \in A) \to (q : x \in A) \to A - p \sim A - q\\
&\mathit{trade}\ (\mathit{here})\ \ (\mathit{here}) \ \ = \iota\\
&\mathit{trade}\ (\mathit{inl}\ p)\ (\mathit{inl}\ q) \ = \mathit{trade}\ p\ q \uplus \iota\\
&\mathit{trade}\ (\mathit{inl}\ p)\ (\mathit{inr}\ q) \ = (\iota \uplus \mathit{to\text{-}rear}\ q)\ ;\ \sigma\ \mathit{paws}\ ;\ (\sigma\ (\mathit{to\text{-}rear}\ p) \uplus \iota)
\end{aligned}$$

```
trade (inr p) (inl q)  =  (to-rear q ⊎ ι) ⨾ paws ⨾ (ι ⊎ σ (to-rear p))
trade (inr p) (inr q)  =  ι ⊎ trade p q
```

While the paths agree, we recursively invoke *trade*. If the paths diverge, we resort to *to-rear*. Consider the third equation. The proof involves three rewrites:

$$\begin{aligned}&(A_1 - p) \uplus A_2 \;\sim\; (A_1 - p) \uplus ((A_2 - q) \uplus \Lbag\, x \,\Rbag)\\ &\quad\sim\; ((A_1 - p) \uplus \Lbag\, x \,\Rbag) \uplus (A_2 - q) \;\sim\; A_1 \uplus (A_2 - q)\end{aligned}$$

We first move the element to the rear in the right subtree, *to-rear q*, then we shift it over to the left, *σ paws*, so that we can finally undo the split, *σ (to-rear p)*.

Now that all the necessary prerequisites are in place, we can finally discharge a proof obligation: antisymmetry of the subbag relation. Recall that $A \subseteq B$ means that there is a "delta bag" Δ such that $A \uplus \Delta \sim B$. The proof strategy is probably clear: if both $A \subseteq B$ and $B \subseteq A$ hold, then both deltas must be equivalent to the empty bag. Using the assumptions, we first show:

$$A \uplus (\Delta_1 \uplus \Delta_2) \;\sim\; (A \uplus \Delta_1) \uplus \Delta_2 \;\sim\; B \uplus \Delta_2 \;\sim\; A \;\sim\; A \uplus \Lbag\Rbag$$

Now we can cancel A and invoke conicality. The rest is routine:

```
⊆-antisymmetric : A ⊆ B → B ⊆ A → A ∼ B
⊆-antisymmetric (Δ₁ such-that ϕ₁) (Δ₂ such-that ϕ₂)  =
  let Δ₁⊎Δ₂∼⟅⟆  =  cancel-left (σ α ⨾ ϕ₁ ⊎ ι ⨾ ϕ₂ ⨾ σ ρ)
      Δ₁∼⟅⟆      =  conical Δ₁⊎Δ₂∼⟅⟆
  in σ ρ ⨾ ι ⊎ σ Δ₁∼⟅⟆ ⨾ ϕ₁
```

We have mentioned before that other free structures, lists and finite sets, can be implemented in a similar way, simply by dropping or adding axioms. Are these free structures cancellative, as well? Yes and no. Lists clearly are: $A +\!\!\!+ B \sim A +\!\!\!+ C$ implies $B \sim C$. (The proof above uses commutativity, see *to-rear*, but not in an essential way, so it can be adapted to work with lists.) Finite sets, however, are not cancellative: $\{a\} \cup \{a\} \sim \{a\} \cup \{\}$, but it is not the case that $\{a\} \sim \{\}$. (The proof above cannot be extended to work with idempotency.)

7 DECIDABILITY

We have come pretty far without posing any restrictive assumptions on the element type such as decidable equality.

```
_≡?_ : (a b : X) → Decide (a ≡ b)
```

However, for some operations we need to assume decidable equality for principled reasons. Here is why. Equality is closely tied to a variety of bag operations and relations.

$$\begin{aligned} x \equiv y &\;\leftrightarrow\; \Lbag\, x \,\Rbag \setminus \Lbag\, y \,\Rbag \sim \Lbag\Rbag\\ x \equiv y &\;\leftrightarrow\; x \,\#\, \Lbag\, y \,\Rbag \equiv 1\\ x \equiv y &\;\leftrightarrow\; x \in \Lbag\, y \,\Rbag\\ x \equiv y &\;\leftrightarrow\; \Lbag\, x \,\Rbag \sim \Lbag\, y \,\Rbag \end{aligned}$$

Consider, for example, bag difference. Using the first characterisation we can reduce equality to the test for emptyness, which is decidable. Consequently, the implementation of difference must involve an equality test. Similar arguments apply to the multiplicity function, written $x \,\#\, A$, to the test for membership, written $x \in?\, A$, and to the test for equivalence, written $A \sim?\, B$. This shows, in particular, that bag implementations based on multiplicity necessarily require decidable equality.

Now, performing a blind search we can decide membership.

```
_∈?_ : (x : X) → (A : Bag X) → Decide (x ∈ A)
x ∈? ⟅ a ⟆ with x ≡? a
... | yes reflexive  =  yes here
... | no ¬x≡a        =  no (λ where here → ¬x≡a reflexive)
x ∈? ⟅⟆              =  no (λ ())
x ∈? (A ⊎ B) with x ∈? A
... | yes x∈A        =  yes (inl x∈A)
... | no ¬x∈A with x ∈? B
... | yes x∈B        =  yes (inr x∈B)
... | no ¬x∈B        =  no (λ where
                           (inl x∈A) → ¬x∈A x∈A
                           (inr x∈B) → ¬x∈B x∈B)
```

Since there is no notion of "negative" occurrences, bag sum $\_ \uplus P$ has no inverse. (You may know the old math joke: "There are 3 mathematicians in a room; 5 leave the room. How many mathematicians must enter the room for it to be empty?") Bag sum has, however, a left adjoint, bag difference, indirectly defined by

$$A \setminus P \subseteq B \;\;\leftrightarrow\;\; A \subseteq B \uplus P$$

The equivalence establishes a Galois connection: $\_\setminus P$ is left adjoint to $\_\uplus P$. From the indirect definition we can systematically *derive* the following implementation of bag difference, which is defined by induction over the structure of the subtrahend.

```
_\_ : Bag X → Bag X → Bag X
A \ ⟅ x ⟆ with x ∈? A
... | yes x∈A   =  A − x∈A
... | no ¬x∈A   =  A
A \ ⟅⟆          =  A
A \ (B₁ ⊎ B₂)   =  (A \ B₂) \ B₁
```

In the singleton case, we test whether x is contained in A. If yes, "bag minus" removes the computed occurrence.

There are, at least, three CM-homomorphisms from the bag monoid into the monoid of natural numbers with addition: cardinality (see Section 5), summing the elements of a bag (not shown), and the multiplicity of a given element (defined below).

```
_#_ : (x : X) → (A : Bag X) → ℕ
x # ⟅ a ⟆ with x ≡? a
... | yes x≡a   =  1
... | no ¬x≡a   =  0
x # ⟅⟆          =  0
x # (A ⊎ B)  =  (x # A) + (x # B)
```

This defines actually a family of CM-homomorphisms, $x \,\#\_$ is a CM-homomorphism for each choice of x.

Quite reassuringly, the properties listed in Remark 1 can be established without too much effort.

$$A \subseteq B \;\leftrightarrow\; (\forall\, x \to x \,\#\, A \preccurlyeq x \,\#\, B) \tag{3}$$

$$A \sim B \;\leftrightarrow\; (\forall\, x \to x \,\#\, A \equiv x \,\#\, B) \tag{4}$$

$$x \,\#\, (A \uplus B) \;\equiv\; (x \,\#\, A) + (x \,\#\, B) \tag{5}$$

$$x \,\#\, (A \setminus B) \;\equiv\; (x \,\#\, A) \mathbin{\dot{-}} (x \,\#\, B) \tag{6}$$

At the risk of dwelling on the obvious, we could, in principle, use the characterisations of $A \subseteq B$ and $A \sim B$ as definitions, but this would come with a loss of generality as we would need to assume decidable equality right from the start (see also Remark 1).

The attentive reader will not have failed to notice that min-intersection and max-union are missing in the list above. Their definitions need some additional machinery to be introduced next.

8 VIEWS

Occasionally it is useful to recurse over the cardinality of a bag. To this end, we provide the following view:

```
data View (A : Bag X) : Set where
  Empty : A ∼ ⟅⟆ → View A
  Laden : (a : X) → (A′ : Bag X) → A ∼ ⟅ a ⟆ ⊎ A′ → View A
```

Under this view, a bag A is either (1) empty: *Empty* ϕ records that A equivales the empty bag; or (2) non-empty: *Laden* a A' ϕ splits the given bag into an element a and a residual bag A' such that $A \sim$ ⟅ a ⟆ ⊎ A'. In brief, the view allows us to select an element from a non-empty bag. The following implementation rather arbitrarily picks the leftmost element, if any.

```
select : (A : Bag X) → View A
select ⟅ x ⟆                      = Laden x ⟅⟆ (σ ρ)
select ⟅⟆                         = Empty ι
select (A ⊎ B) with select A
select (A ⊎ B) | Empty φ with select B
... | Empty ψ                     = Empty (φ ⊎ ψ ; ƛ)
... | Laden b B′ ψ                = Laden b B′ (φ ⊎ ψ ; ƛ)
select (A ⊎ B) | Laden a A′ φ     = Laden a (A′ ⊎ B) ((φ ⊎ ι) ; α)
```

To illustrate the use of the view, let us implement one of the remaining "set-like" operations on bags: min-intersection. The operation is defined by well-founded recursion on _≺_, the standard strict ordering on the natural numbers. Well-founded recursion can, for example, be realised using an accessibility predicate [9].

```
data Accessible (n : ℕ) : Set where
  access : (∀ {m} → m ≺ n → Accessible m) → Accessible n
```

The implementation of min-intersection consists of a worker, called *intersection*, and a wrapper, the actual operation _∩_.

```
intersect : (A B : Bag X) → Accessible (card A) → Bag X
intersect A B (access since) with select A
... | Empty φ  = ⟅⟆
... | Laden a A′ φ with a ∈? B
... | yes a∈B   = ⟅ a ⟆ ⊎ intersect A′ (B − a∈B) (since (card-≻ φ))
... | no ¬a∈B   =         intersect A′ B          (since (card-≻ φ))
_∩_ : Bag X → Bag X → Bag X
A ∩ B  =  intersect A B well-founded
```

The worker takes an additional argument of type *Accessible* (*card* A) that drives the recursion. Each recursive call is obliged to prove that the cardinality of its first argument decreases. As a convenience, the library defines:

```
card-≻ : A ∼ ⟅ a ⟆ ⊎ A′ → card A′ ≺ card A
```

The wrapper finally calls the worker, providing evidence that the natural numbers are well-founded, that each natural number is accessible, *well-founded* : ∀ {n : ℕ} → *Accessible* n.

Intersection works by repeatedly calling *select*, which realises a "cons" view of bags. Of course, the library wouldn't be feature-complete without providing a function that encapsulates this recursion pattern, providing a recursive "list" view.

```
data List-View (A : Bag X) : Set where
  []    :  {φ : A ∼ ⟅⟆} → List-View A
  _::_  :  (a : X) → {A′ : Bag X} → {φ : A ∼ ⟅ a ⟆ ⊎ A′}
           → List-View A′ → List-View A
```

To reduce clutter, the equivalence proofs and the residual bag are now implicit arguments, indicated by curly braces. Like intersection, the actual view function consists of a worker and a wrapper.

```
list-view′ : (A : Bag X) → Accessible (card A) → List-View A
list-view′ A (access since) with select A
... | Empty φ        = [] {φ = φ}
... | Laden a A′ φ  =  _::_ a {φ = φ} (list-view′ A′ (since (card-≻ φ)))
list-view : (A : Bag X) → List-View A
list-view A  =  list-view′ A well-founded
```

For variety, we use the list view to implement max-union — min-intersection can be rewritten accordingly.

```
union : {A : Bag X} → List-View A → (B : Bag X) → Bag X
union []          B  =  B
union (a :: A′) B with a ∈? B
... | yes a∈B      = ⟅ a ⟆ ⊎ union A′ (B − a∈B)
... | no ¬a∈B      = ⟅ a ⟆ ⊎ union A′ B
_∪_ : Bag X → Bag X → Bag X
A ∪ B  =  union (list-view A) B
```

The tests for equivalence A ∼? B and containment A ⊆? B can be be realised using similar definitions.

Turning to properties, the following laws explain the prefixes "min" and "max" (see also Remark 1). They can be shown using fairly straightforward inductive proofs.

$$x \mathbin{\#} (A \cap B) \equiv (x \mathbin{\#} A) \downarrow (x \mathbin{\#} B)$$
$$x \mathbin{\#} (A \cup B) \equiv (x \mathbin{\#} A) \uparrow (x \mathbin{\#} B)$$

The characterisations in terms of multiplicities are instrumental for establishing laws, such as,

$$A \cap B \subseteq A \cup B \subseteq A \uplus B$$

which follows from $a \downarrow b \preccurlyeq a \uparrow b \preccurlyeq a + b$. In particular, one can show that bags form a distributive lattice taking min-intersection as meet and max-union as join. The proof is essentially based on the fact that the natural numbers with minimum and maximum form a distributive lattice.

9 RELATED WORK

Our work is closest in spirit to Bird's "Lectures on Constructive Functional Programming" [3], which introduce a calculus for deriving functional programs from their specification. In particular, Bird suggests a common notation for lists, bags, and sets, which has to become known as *join lists*. In some sense, our *Bag* datatype materialises his notation.[2] And, of course, we could follow Bird's lead and use the *same* datatype for trees, lists, bags, and sets, simply by endowing it with different equivalence relations.

Approaches based on multiplicity. The Coq standard library [13] defines bags over X as functions of type $X \to \mathbb{N}$. An approach based on multiplicities, but now with finite maps $X \to_{\mathrm{fin}} \mathbb{N}$, is also used by Angiuli et al. in one of their examples. As pointed out in Remarks 1 and 2, these approaches depend fundamentally on decidable equality, which makes them less widely applicable.

Approaches based on sequence types. We have noted that there are two degrees of freedom when it comes to implementing bags based on sequence types: the type itself and the notion of equivalence. Several combinations have been presented and implemented in other publications, repositories, or by ourselves. The following table shows how the 2-dimensional design space is populated.

	Fin n → *X*	cons lists	join lists
multiplicity (4)		[1]	*derived*, see (4)
membership (2)		[8], [14]	*derived*, see §4
permutations	[5]	([8])	
proof trees			*we are here*

It is important that the equivalence on bags supports element types with user-defined equality. Equivalence based on proof-relevant membership cannot afford this. In general, evidence for $x \in A$ consists of a "path" in A *and* a proof that x is equal to the element to which the path leads. The problem is that proof-relevance of the equality type now matters. In other words, an equivalence based on proof-relevant membership does not only compare paths but also the number of equality proofs. Therefore, the approach works only in special cases, e.g. for propositional equality or when equality on the element type is actually proof-irrelevant.

The approaches based on proof-relevant membership and on permutations are defined via bijections. Hence, we have to deal with properties concerning functions which is often inconvenient (see Remark 3). By contrast, working with inductively defined proof trees is a lot smoother, as Agda is tailored towards that purpose. Loosely speaking, there is a difference in the underlying philosophies. The former approaches are piecemeal, focusing on individual elements and path transformations, while the proof tree approach is holistic, considering the entire tree and tree transformations.

Approaches based on quotient types. The bag type and the equivalence type can be merged into a higher inductive type (HIT) in Cubical Agda [15]:

```
data Bag (X : Set) : Set where
  ≀_ʃ : X → Bag X
  ≀ʃ  : Bag X
  _⊎_ : Bag X → Bag X → Bag X
  λ : {A     : Bag X} → ≀ʃ ⊎ A ≡ A
  γ : {A B   : Bag X} → A ⊎ B ≡ B ⊎ A
  α : {A B C : Bag X} → (A ⊎ B) ⊎ C ≡ A ⊎ (B ⊎ C)
  truncation : ∀ (A B : Bag X) → (p q : A ≡ B) → p ≡ q
```

[2]On a historical note, the notation for bag brackets, ≀ and ʃ, was designed by the Programming Research Group at Oxford (personal communication with Jeremy Gibbons).

Observe that equivalence and congruence axioms are not needed, which simplifies equational reasoning about bags and defining functions from bags. The *truncation* axiom enforces that equality proofs of the same type are themselves equal — loosely speaking, equality proofs are unique. This technicality is generally required for quotient types.

Generally, bag types as HITs can be defined by taking an arbitrary, inductively defined sequence type, adding appropriate path constructors for equalities. For example, the Agda Cubical standard library [15] and Pitts [11] define bags as lists with the equality:

$a :: b :: as \equiv b :: a :: as$

Choudhury and Fiore [6] use the same underlying sequence type but use a minor variation of the law above:

$as \equiv b :: cs \to a :: cs \equiv bs \to a :: as \equiv b :: bs$

Overall, the design choices boil down to the ones shown below.

	cons lists	join lists
proof trees	[6], [15], [11]	[2], [7], [15]

Without a doubt, higher inductive types are attractive from a theoretical point of view. But it remains to be seen whether there are also practical benefits in terms of proof efficiency and perspicuity.

10 CONCLUSION

When we teach functional programming we never tire of repeating the following mantra: "The basic building blocks of functional programs are type declarations — a type describes data — and function definitions — a function operates on data." Inductive datatypes are at the heart of functional programming (not the λ-calculus), enabling an attractive equational style based on pattern matching and recursion. The lesson we have learned is that we better stick to our own advice, especially in a dependently typed setting, where evidence is data and proofs are programs.

For a course on program verification, we tried several implementations of bags. The one that worked best and is reported on in this paper is based on inductive datatypes. Bags are given by bag expressions quotiented by laws of commutative monoids. The syntax is described by a datatype, as is the evidence that two expressions are equivalent. The approach has both theoretical and practical merits. The requirements on the meta-theory and on the element type are minimal. Agda's proof assistant is tailored to inductive datatypes: the goals (e.g. for equivalence proofs) are short and clear, proof automation (Agsy) works reasonably well. As a further bonus, as we deal with expression *trees* and proofs *trees*, programs and proofs have a nice algorithmic touch.

There is still work to be done. For the purposes of presentation, we have made the simplifying assumption that the equivalence on the element type is given by propositional equality (see Section 4). Of course, a bag library based on this assumption is not going to fly. The good news is that the approach can be adapted to work with an arbitrary underlying equivalence relation — some adjustments are required here and there. What remains is large an engineering issue: How to proceed in practical terms, should we switch to setoids or employ "type classes", based on instance declarations and instance arguments (see Section 5)? We hope to be able to report on our experiences in the not too distant future.

ACKNOWLEDGEMENTS

We wish to thank Jacques Carette for pointing out an error in an earlier version of this paper and for indicating relevant references.

REFERENCES

[1] Carlo Angiuli, Evan Cavallo, Anders Mörtberg, and Max Zeuner. 2021. Internalizing Representation Independence with Univalence. *Proc. ACM Program. Lang.* 5, POPL, Article 12 (jan 2021). https://doi.org/10.1145/3434293

[2] Henning Basold, Herman Geuvers, and Niels van der Weide. 2017. Higher inductive types in programming. *Journal of Universal Computer Science* 23, 1 (2017), 63–88.

[3] Richard S. Bird. 1988. Lectures on Constructive Functional Programming. In *Constructive Methods in Computer Science*, Manfred Broy (Ed.). Springer-Verlag.

[4] Ana Bove, Peter Dybjer, and Ulf Norell. 2009. A Brief Overview of Agda - A Functional Language with Dependent Types. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5674)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer, 73–78. https://doi.org/10.1007/978-3-642-03359-9_6

[5] Jacques Carette, Musa Al-hassy, and Wolfram Kahl. 2018. A tale of theories and data-structures. https://wiki.hh.se/wg211/images/9/94/M18Carette-Slides.pdf

[6] Vikraman Choudhury and Marcelo Fiore. 2019. The finite-multiset construction in HoTT.

[7] Vikraman Choudhury and Marcelo Fiore. 2023. Free Commutative Monoids in Homotopy Type Theory. *Electronic Notes in Theoretical Informatics and Computer Science* (2023).

[8] Nils Anders Danielsson. 2012. Bag equivalence via a proof-relevant membership relation. In *Interactive Theorem Proving: Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings 3*. Springer, 149–165.

[9] Bengt Nordström. 1988. Terminating General Recursion. *BIT* 28, 3 (sep 1988), 605–619. https://doi.org/10.1007/BF01941137

[10] L. C. Paulson. 1996. *ML for the Working Programmer* (2nd ed.). Cambridge University Press.

[11] Andrew M Pitts. 2020. Quotients in Dependent Type Theory. In *5th International Conference on Formal Structures for Computation and Deduction*. https://www.cl.cam.ac.uk/~amp12/talks/FSCD2020-s3-slides.pdf

[12] Aaron Stump. 2016. *Verified Functional Programming in Agda*. Association for Computing Machinery and Morgan & Claypool.

[13] The Coq Development Team. 2022. *The Coq Proof Assistant*. https://doi.org/10.5281/zenodo.7313584

[14] The Agda Community. 2023. *Agda Standard Library*. https://github.com/agda/agda-stdlib

[15] The agda/cubical development team. 2018–2023. The agda/cubical library. https://github.com/agda/cubical/

[16] Twan van Laarhoven. [n. d.]. The complete correctness of sorting. https://www.twanvl.nl/blog/agda/sorting