A tale of theories and data-structures

Jacques Carette, Musa Al-hassy, Wolfram Kahl

McMaster University, Hamilton

June 4, 2018

Lists and Monoids

Claim

A List is a Free Monoid

What does that really mean?

Lists and Monoids

Claim

A List is a Free Monoid

What does that really mean?

Fancy explanation: The functor from the category Types of types and function, with `List` as its object mapping and `map` for homomorphism, to the category Monoid of monoids and monoid homomorphisms, is left adjoint to the forgetful functor (from Monoid to Types).

Lists and Monoids

Claim

A List is a Free Monoid

What does that really mean?

Fancy explanation: The functor from the category Types of types and function, with `List` as its object mapping and `map` for homomorphism, to the category Monoid of monoids and monoid homomorphisms, is left adjoint to the forgetful functor (from Monoid to Types).

`List` (equipped with constructors `[]`, `::` and functions `map`, `++`, `singleton`, and `foldr`) is the language of monoids. In other words, `List` is the canonical term syntax for computing with monoids.

Lists and Monoids

Claim

A List is a Free Monoid

What does that really mean?

Fancy explanation: The functor from the category Types of types and function, with `List` as its object mapping and `map` for homomorphism, to the category Monoid of monoids and monoid homomorphisms, is left adjoint to the forgetful functor (from Monoid to Types).

`List` (equipped with constructors `[]`, `::` and functions `map`, `++`, `singleton`, and `foldr`) is the language of monoids. In other words, `List` is the canonical term syntax for computing with monoids.

Why on earth would we care about that? Let's see!

Non-categorical version

The requirements roughly translate to
Monoid:

- Need a *container* C of α
- with a distinguished container e devoid of α's
- a binary operation $*$ that puts two containers together
- such that e is a left/right unit for $*$.

Functor:

- A way to apply a $(\alpha \to \beta)$ function to a $C\,\alpha$ to get a $C\,\beta$
- which "plays well" with $\mathsf{id}, \circ, \equiv$ and $*$.

Adjunction:

- An operation `singleton` embedding an α as a container $C\,\alpha$
- an operation `foldr` (over arbitrary Monoid)
- such that both operations "play well" with each other.

Extremely handy:

- Induction principle

The plot thickens

Given an arbitrary type A :

Theory	**Free Structure**	CoFree
Carrier	Identity A	Identity A
Pointed	Maybe A	–
Unary	Eventually A, $\mathbb{N} \times A$	?
Involutive	$A \uplus A$	A $\times$ A
Magma	Tree A	?
Semigroup	NEList A	?
Monoid	List A	?
Left Unital Semigroup	List $A \times \mathbb{N}$	?
Right Unital Semigroup	$\mathbb{N}\times$ List A	?

The plot thickens

Given an arbitrary type A :

Theory	**Free Structure**	CoFree
Carrier	Identity A	Identity A
Pointed	Maybe A	–
Unary	Eventually A, $\mathbb{N} \times A$	?
Involutive	$A \uplus A$	A × A
Magma	Tree A	?
Semigroup	NEList A	?
Monoid	List A	?
Left Unital Semigroup	List $A \times \mathbb{N}$	?
Right Unital Semigroup	$\mathbb{N}\times$ List A	?

What is the Free Structure? It is “the” **term language in normal form** associated to the theory.

Benefits

Benefits of the formal approach:

- Obvious: Dispell silly conjectures/errors
- Discover some neat relationships between algebraic theories and data-structures
- `fold` (aka the counit)
- Induction

Benefits

Benefits of the formal approach:

- Obvious: Dispell silly conjectures/errors
- Discover some neat relationships between algebraic theories and data-structures
- `fold` (aka the counit)
- Induction

Examples: counit for Unary, Involutive

Extending the tale

Given an arbitrary type A :

Theory	**Free Structure**
Carrier	Identity A
Pointed	Maybe A
Unary	$\mathbb{N} \times A$
Involutive	$A \uplus A$
Magma	Tree A
Semigroup	NEList A
Monoid	List A
Left Unital Semigroup	List $A \times \mathbb{N}$
Right Unital Semigroup	$\mathbb{N}\times$ List A
Commutative Monoid	?
Group	?
Abelian Group	?
Idempotent Comm. Monoid	?

Commutative Monoid and Bag

Definition

A *Bag* (over a type A) is an unordered finite collection of x where $x : A$.

Commutative Monoid and Bag

Definition

A *Bag* (over a type A) is an unordered finite collection of x where $x : A$.

Implementation?

- Inductive type

Commutative Monoid and Bag

Definition

A *Bag* (over a type A) is an unordered finite collection of x where $x : A$.

Implementation?

- Inductive type
  - Ordered!

Commutative Monoid and Bag

Definition

A *Bag* (over a type A) is an unordered finite collection of x where $x : A$.

Implementation?

- Inductive type
  - Ordered!
- $A \to \mathbb{N}$

Commutative Monoid and Bag

Definition

A *Bag* (over a type A) is an unordered finite collection of x where $x : A$.

Implementation?

- Inductive type
  - Ordered!
- $A \to \mathbb{N}$
  - No finite support!

Commutative Monoid and Bag

Definition

A *Bag* (over a type A) is an unordered finite collection of x where $x : A$.

Implementation?

- Inductive type
  - Ordered!
- $A \to \mathbb{N}$
  - No finite support!
- $A \to \mathbb{N}$ plus finite support

Commutative Monoid and Bag

Definition

A *Bag* (over a type A) is an unordered finite collection of x where $x : A$.

Implementation?

- Inductive type
  - Ordered!
- $A \to \mathbb{N}$
  - No finite support!
- $A \to \mathbb{N}$ plus finite support
  - "Finite support" is hard to say constructively . . .

Commutative Monoid and Bag

Definition

A *Bag* (over a type A) is an unordered finite collection of x where $x : A$.

Implementation?

- Inductive type
  - Ordered!
- $A \to \mathbb{N}$
  - No finite support!
- $A \to \mathbb{N}$ plus finite support
  - "Finite support" is hard to say constructively ...
  - Summing over all elements of A is even harder ...

Commutative Monoid and Bag

Definition

A *Bag* (over a type A) is an unordered finite collection of x where $x : A$.

Implementation?

- Inductive type
  - Ordered!
- $A \to \mathbb{N}$
  - No finite support!
- $A \to \mathbb{N}$ plus finite support
  - "Finite support" is hard to say constructively ...
  - Summing over all elements of A is even harder ...
  - Can build a decidable equiv. relation on A from $A \to \mathbb{N}$!

Commutative Monoid and Bag

Definition

A *Bag* (over a type A) is an unordered finite collection of x where $x : A$.

Implementation?

- Inductive type
  - Ordered!
- $A \to \mathbb{N}$
  - No finite support!
- $A \to \mathbb{N}$ plus finite support
  - "Finite support" is hard to say constructively ...
  - Summing over all elements of A is even harder ...
  - Can build a decidable equiv. relation on A from $A \to \mathbb{N}$!
- `List` A up to bag-equality (aka permutations)

Commutative Monoid and Bag

Definition

A *Bag* (over a type A) is an unordered finite collection of x where $x : A$.

Implementation?

- Inductive type
  - Ordered!
- $A \to \mathbb{N}$
  - No finite support!
- $A \to \mathbb{N}$ plus finite support
  - "Finite support" is hard to say constructively …
  - Summing over all elements of A is even harder …
  - Can build a decidable equiv. relation on A from $A \to \mathbb{N}$!
- `List` A up to bag-equality (aka permutations)
  - almost works!

Commutative Monoid and Bag

Definition

A *Bag* (over a type A) is an unordered finite collection of x where $x : A$.

Implementation?

- Inductive type
  - Ordered!
- $A \to \mathbb{N}$
  - No finite support!
- $A \to \mathbb{N}$ plus finite support
  - "Finite support" is hard to say constructively ...
  - Summing over all elements of A is even harder ...
  - Can build a decidable equiv. relation on A from $A \to \mathbb{N}$!
- `List` A up to bag-equality (aka permutations)
  - almost works!
  - Commutative Monoid uses $\equiv$

Commutative Monoid and Bag

Definition

A *Bag* (over a type A) is an unordered finite collection of x where $x : A$.

Implementation?

- Inductive type
  - Ordered!
- $A \to \mathbb{N}$
  - No finite support!
- $A \to \mathbb{N}$ plus finite support
  - "Finite support" is hard to say constructively ...
  - Summing over all elements of A is even harder ...
  - Can build a decidable equiv. relation on A from $A \to \mathbb{N}$!
- `List` A up to bag-equality (aka permutations)
  - almost works!
  - Commutative Monoid uses $\equiv$

Theorem (Within Martin-Löf Type Theory)

There's no free functor from Types to Commutative Monoids using $\equiv$.

Change the question!

Definition

A *DBag* over a type A with dec. $=$ is an unordered collection of x where $x : A$.

Change the question!

Definition

A *DBag* over a type A with dec. $=$ is an unordered collection of x where $x : A$.

Definition

A *Bag* over a setoid A is an unordered collection of x where $x : \texttt{Carrier}\ A$.

Change the question!

Definition

A *DBag* over a type A with dec. $=$ is an unordered collection of x where $x \ : \ A$.

Definition

A *Bag* over a setoid A is an unordered collection of x where $x : \texttt{Carrier}\, A$.

Implementation attempts:

Change the question!

Definition

A *DBag* over a type A with dec. $=$ is an unordered collection of x where $x : A$.

Definition

A *Bag* over a setoid A is an unordered collection of x where x : `Carrier` A.

Implementation attempts:

- Nils Anders Danielsson's *Bag Equivalence via a Proof-Relevant Membership Relation*

Change the question!

Definition

A *DBag* over a type A with dec. $=$ is an unordered collection of x where $x \; : \; A$.

Definition

A *Bag* over a setoid A is an unordered collection of x where $x : \texttt{Carrier}\ A$.

Implementation attempts:

- Nils Anders Danielsson's *Bag Equivalence via a Proof-Relevant Membership Relation*
  - Too many parts over $\equiv$

Change the question!

Definition

A *DBag* over a type A with dec. $=$ is an unordered collection of x where $x : A$.

Definition

A *Bag* over a setoid A is an unordered collection of x where x : `Carrier` A.

Implementation attempts:

- Nils Anders Danielsson's *Bag Equivalence via a Proof-Relevant Membership Relation*
  - Too many parts over $\equiv$
- Erik Palmgren's *Setoid Families*

Change the question!

Definition

A *DBag* over a type A with dec. $=$ is an unordered collection of x where $x \;:\; A$.

Definition

A *Bag* over a setoid A is an unordered collection of x where $x : \texttt{Carrier}\ A$.

Implementation attempts:

- Nils Anders Danielsson's *Bag Equivalence via a Proof-Relevant Membership Relation*
  - Too many parts over $\equiv$
- Erik Palmgren's *Setoid Families*
  - Extremely complex, forget the actual dead end.

Change the question!

Definition

A *DBag* over a type A with dec. $=$ is an unordered collection of x where $x\ :\ A$.

Definition

A *Bag* over a setoid A is an unordered collection of x where $x : \texttt{Carrier}\ A$.

Implementation attempts:

- Nils Anders Danielsson's *Bag Equivalence via a Proof-Relevant Membership Relation*
  - Too many parts over $\equiv$
- Erik Palmgren's *Setoid Families*
  - Extremely complex, forget the actual dead end.
- Mimick above with our own Proof-Relevant $\in$ over Setoid

Change the question!

Definition

A *DBag* over a type A with dec. $=$ is an unordered collection of x where $x \ : \ A$.

Definition

A *Bag* over a setoid A is an unordered collection of x where x : `Carrier` A.

Implementation attempts:

- Nils Anders Danielsson's *Bag Equivalence via a Proof-Relevant Membership Relation*
  - Too many parts over $\equiv$
- Erik Palmgren's *Setoid Families*
  - Extremely complex, forget the actual dead end.
- Mimick above with our own Proof-Relevant $\in$ over Setoid
  - Proof that `fold` well-behaved very hard.

Change the question!

Definition

A *DBag* over a type A with dec. $=$ is an unordered collection of x where $x \ : \ A$.

Definition

A *Bag* over a setoid A is an unordered collection of x where x : `Carrier` A.

Implementation attempts:

- Nils Anders Danielsson's *Bag Equivalence via a Proof-Relevant Membership Relation*
  - Too many parts over $\equiv$
- Erik Palmgren's *Setoid Families*
  - Extremely complex, forget the actual dead end.
- Mimick above with our own Proof-Relevant $\in$ over Setoid
  - Proof that `fold` well-behaved very hard.
- Bag-equality in new version of Agda!

Change the question!

Definition

A *DBag* over a type A with dec. $=$ is an unordered collection of x where $x \; : \; A$.

Definition

A *Bag* over a setoid A is an unordered collection of x where x : `Carrier` A.

Implementation attempts:

- Nils Anders Danielsson's *Bag Equivalence via a Proof-Relevant Membership Relation*
  - Too many parts over $\equiv$
- Erik Palmgren's *Setoid Families*
  - Extremely complex, forget the actual dead end.
- Mimick above with our own Proof-Relevant $\in$ over Setoid
  - Proof that `fold` well-behaved very hard.
- Bag-equality in new version of Agda!
  - Still assumes $\equiv$.

Change the question!

Definition

A *DBag* over a type A with dec. $=$ is an unordered collection of x where $x : A$.

Definition

A *Bag* over a setoid A is an unordered collection of x where x : `Carrier` A.

Implementation attempts:

- Nils Anders Danielsson's *Bag Equivalence via a Proof-Relevant Membership Relation*
  - Too many parts over $\equiv$
- Erik Palmgren's *Setoid Families*
  - Extremely complex, forget the actual dead end.
- Mimick above with our own Proof-Relevant $\in$ over Setoid
  - Proof that `fold` well-behaved very hard.
- Bag-equality in new version of Agda!
  - Still assumes $\equiv$.
- Experimental library with permutations over tables
  $\Rightarrow$ proof that `fold` is well-behaved

Success!

Key ingredients of Bag

Distilling the insights from $\sim$1000 lines of Agda

Key ingredients of Bag

Distilling the insights from ∼1000 lines of Agda

- Internalize length of “list” into a record `Seq` —`subst` is evil!
- Table of A is `Fin` $\mathbb{N} \to A$ (finite support)

Key ingredients of Bag

Distilling the insights from $\sim$1000 lines of Agda

- Internalize length of “list” into a record `Seq` —`subst` is evil!
- Table of A is `Fin` $\mathbb{N} \to A$ (finite support)
- Equivalence of sequences S and T is

Key ingredients of Bag

Distilling the insights from $\sim$1000 lines of Agda

- Internalize length of "list" into a record `Seq` —`subst` is evil!
- Table of A is `Fin` $\mathbb{N} \to A$ (finite support)
- Equivalence of sequences S and T is
  - A permutation between $|S|$ and $|T|$, i.e.

Key ingredients of Bag

Distilling the insights from $\sim$1000 lines of Agda

- Internalize length of "list" into a record `Seq` —`subst` is evil!
- Table of A is $\texttt{Fin}\,\mathbb{N} \to A$ (finite support)
- Equivalence of sequences S and T is
  - A permutation between $|S|$ and $|T|$, i.e.
  - An equivalence between `Fin` $|S|$ and `Fin` $|T|$

Key ingredients of Bag

Distilling the insights from $\sim$1000 lines of Agda

- Internalize length of "list" into a record `Seq` —`subst` is evil!
- Table of A is $\texttt{Fin}\,\mathbb{N} \to A$ (finite support)
- Equivalence of sequences S and T is
  - A permutation between $|S|$ and $|T|$, i.e.
  - An equivalence between `Fin` $|S|$ and `Fin` $|T|$
  - A proof that permuting the elements of T gives a pointwise Setoid-equivalence to those of S.

Key ingredients of Bag

Distilling the insights from $\sim$1000 lines of Agda

- Internalize length of "list" into a record `Seq` —`subst` is evil!
- Table of A is `Fin` $\mathbb{N} \to A$ (finite support)
- Equivalence of sequences S and T is
  - A permutation between $|S|$ and $|T|$, i.e.
  - An equivalence between `Fin` $|S|$ and `Fin` $|T|$
  - A proof that permuting the elements of T gives a pointwise Setoid-equivalence to those of S.
- Use previous infrastructure built to move between proofs on permutations and proofs on types (work on Π languages w/ Amr Sabry)

Key ingredients of Bag

Distilling the insights from $\sim$1000 lines of Agda

- Internalize length of “list” into a record `Seq` —`subst` is evil!
- Table of A is `Fin` $\mathbb{N} \to A$ (finite support)
- Equivalence of sequences S and T is
  - A permutation between $|S|$ and $|T|$, i.e.
  - An equivalence between `Fin` $|S|$ and `Fin` $|T|$
  - A proof that permuting the elements of T gives a pointwise Setoid-equivalence to those of S.
- Use previous infrastructure built to move between proofs on permutations and proofs on types (work on Π languages w/ Amr Sabry)
- Create an abstract interface for Multiset, MultisetHom and “functorial” MultisetHom

Key ingredients of Bag

Distilling the insights from $\sim$1000 lines of Agda

- Internalize length of "list" into a record `Seq` —`subst` is evil!
- Table of A is $\texttt{Fin}\,\mathbb{N} \to A$ (finite support)
- Equivalence of sequences S and T is
  - A permutation between $|S|$ and $|T|$, i.e.
  - An equivalence between `Fin` $|S|$ and `Fin` $|T|$
  - A proof that permuting the elements of T gives a pointwise Setoid-equivalence to those of S.
- Use previous infrastructure built to move between proofs on permutations and proofs on types (work on Π languages w/ Amr Sabry)
- Create an abstract interface for Multiset, MultisetHom and "functorial" MultisetHom
- Satisfies interface $\Rightarrow$ left adjoint to Commutative Monoid

Key ingredients of Bag

Distilling the insights from $\sim$1000 lines of Agda

- Internalize length of "list" into a record `Seq` —`subst` is evil!
- Table of A is $\texttt{Fin}\,\mathbb{N} \to A$ (finite support)
- Equivalence of sequences S and T is
  - A permutation between $|S|$ and $|T|$, i.e.
  - An equivalence between `Fin` $|S|$ and `Fin` $|T|$
  - A proof that permuting the elements of T gives a pointwise Setoid-equivalence to those of S.
- Use previous infrastructure built to move between proofs on permutations and proofs on types (work on Π languages w/ Amr Sabry)
- Create an abstract interface for Multiset, MultisetHom and "functorial" MultisetHom
- Satisfies interface ⇒ left adjoint to Commutative Monoid
- Bag satisfies the interface

Key ingredients of Bag

Distilling the insights from $\sim$1000 lines of Agda

- Internalize length of "list" into a record `Seq` —`subst` is evil!
- Table of A is `Fin` $\mathbb{N} \to A$ (finite support)
- Equivalence of sequences S and T is
  - A permutation between $|S|$ and $|T|$, i.e.
  - An equivalence between `Fin` $|S|$ and `Fin` $|T|$
  - A proof that permuting the elements of T gives a pointwise Setoid-equivalence to those of S.
- Use previous infrastructure built to move between proofs on permutations and proofs on types (work on Π languages w/ Amr Sabry)
- Create an abstract interface for Multiset, MultisetHom and "functorial" MultisetHom
- Satisfies interface $\Rightarrow$ left adjoint to Commutative Monoid
- Bag satisfies the interface
- Use `abstract` in key places to prevent normalization in proof goals

Key ingredients of Bag

Distilling the insights from ∼1000 lines of Agda

- Internalize length of "list" into a record `Seq` —`subst` is evil!
- Table of A is `Fin` $\mathbb{N} \to A$ (finite support)
- Equivalence of sequences S and T is
  - A permutation between $|S|$ and $|T|$, i.e.
  - An equivalence between `Fin` $|S|$ and `Fin` $|T|$
  - A proof that permuting the elements of T gives a pointwise Setoid-equivalence to those of S.
- Use previous infrastructure built to move between proofs on permutations and proofs on types (work on Π languages w/ Amr Sabry)
- Create an abstract interface for Multiset, MultisetHom and "functorial" MultisetHom
- Satisfies interface ⇒ left adjoint to Commutative Monoid
- Bag satisfies the interface
- Use `abstract` in key places to prevent normalization in proof goals
- Never use `subst` —even when building the identity permutation

Extending the tale, take 2

Given an arbitrary type A :

Theory	**Structure**	Over	Equality
Carrier	Identity A	Type	$\equiv$
Pointed	Maybe A	Type	$\equiv$
Unary	$\mathbb{N} \times$ A	Type	$\equiv$
Involutive	A $\uplus$ A	Type	$\equiv$
Magma	Tree A	Type	$\equiv$
Semigroup	NEList A	Type	$\equiv$
Monoid	List A	Type	$\equiv$
Left Unital Semigroup	List A $\times \mathbb{N}$	Type	$\equiv$
Right Unital Semigroup	$\mathbb{N} \times$ List A	Type	$\equiv$
Commutative Monoid	Bag	Setoid	proof-relevant permutations
Group	?	?	?
Abelian Group	Hybrid Sets	Setoid	proof-relevant permutations
Idemp. Comm. Monoid	Set	Setoid	logical equivalence

What's the deal with those axioms?

- Works easily:
  - Associativity: $\forall x, y, z.\ x * (y * z) \equiv (x * y) * z$;
  - Left-unit: $\forall x.\ e * x \equiv x$;
  - Right-unit: $\forall x.\ x * e \equiv x$
  - Involutive: $\forall x.\ inv(inv x) \equiv x$
- Hard:
  - Commutativity: $\forall x, y.\ x * y \equiv y * x$
- Very Hard:
  - Idempotence: $\forall x.\ x * x \equiv x$

What's the deal with those axioms?

- Works easily:
  - Associativity: $\forall x, y, z.\ x * (y * z) \equiv (x * y) * z$;
  - Left-unit: $\forall x.\ e * x \equiv x$;
  - Right-unit: $\forall x.\ x * e \equiv x$
  - Involutive: $\forall x.\ inv(inv x) \equiv x$
- Hard:
  - Commutativity: $\forall x, y.\ x * y \equiv y * x$
- Very Hard:
  - Idempotence: $\forall x.\ x * x \equiv x$

Found the secret ingredient in *Algebraic Theories in Monoidal Categories* by L. Mauri: structural context rules (weakening, exchange, contraction).

More tale to tell

- $\bot$, $\top$, $\mathbb{B}$, $\mathbb{N}$, $\mathbb{Z}$ show up as initial objects.
- Bivariate (but $\times$ and $\uplus$ are adjoint to diagonal, not forgetful functor)
- Indexed sets of operations

Potential data-structures

left-zero monoid, pointed unary, idempotent unary, commutative magma, pointed magma, quasigroup, loop, semilattice, medial magma, left semimedial magma, left distributive magma, idempotent magma, zeropotent magma, left unary magma, Steiner magma, null semigroup, BCI algebra, BCK algebra, squag, sloop, Moufang quasigroup, loop, left shelf, shelf, rack, spindle, quandle, Kei, involutive semigroup, band, rectangular band, hemigroup, pseudo inverse algebra, ringoid, left near semiring, near semiring, semifield, semiring, semirng, pre-dioid, dioid, star semiring, idempotent dioid, ring, commutative ring, idempotent semiring, Stone algebra, Kleene lattice, Kleene algebra, Heyting algebra, Goedel algebra, ortho lattice, directoid, semiheap, idempotent semiheap, heap, meadow, wheel.

Structures looking for a home

Difference list, stack, queue, finite map, rose tree, digraph, multigraph, partitions, oriented cycles, colorings, tri-colorings, hedges, derangements, ballots, commutative parenthesizations, linear order, permutations, even permutations, chains, oriented sets, even sets, octopus, vertebrae.

Math and CS

Given an arbitrary type A :

Theory	**Structure**	Over	Equality
Carrier	Identity A	Type	$\equiv$
Pointed	Maybe A	Type	$\equiv$
Unary	$\mathbb{N} \times$ A	Type	$\equiv$
Involutive	A $\uplus$ A	Type	$\equiv$
Magma	Tree A	Type	$\equiv$
Semigroup	NEList A	Type	$\equiv$
Monoid	List A	Type	$\equiv$
Left Unital Semigroup	List A $\times \mathbb{N}$	Type	$\equiv$
Right Unital Semigroup	$\mathbb{N} \times$ List A	Type	$\equiv$
Commutative Monoid	Bag	Setoid	proof-relevant permutations
Group	?	?	?
Abelian Group	Hybrid Sets	Setoid	proof-relevant permutations
Idemp. Comm. Monoid	Set	Setoid	logical equivalence

`https://github.com/JacquesCarette/TheoriesAndDataStructures`