Theories and Data Structures

—Draft—

MUSA AL-HASSY, JACQUES CARETTE, WOLFRAM KAHL

Abstract

The ubiquitous data structures found in computing are accompanied by a core interface for their manipulation. Irrespective of the language, this core interface arises naturally.

With a bit of elementary mathematics, we demonstrate that this is no accident: Data structures provide abstract syntax trees corresponding to meaningful conceptual theories and the coherency of the interface is nothing more than the structure abiding by the laws of the theory.

1 INTRODUCTION

Our story begins with abstract syntax trees.

Programmers write code, which they may want to execute remotely and so they serialise it in a coherent format, transmit it, then have it re-interpreted as meaningful code and executed. We will demonstrate some concepts in an ambient language, discuss what it means for them to be coherent, show that their associated 'free' structures provide a means to serialise them, then demonstrate that the proof obligations for 'free' necessitate an interpretation, or compilation, function.

The essence of our exposition is the adjunction, which requires understanding naturality, functoriality, and categories. In section 2, we review these basic concepts to make this work accessible; then we illustrate the process of obtaining data structures from theories by considering adjunctions. Section 3 demonstrates this process by obtaining concepts such as for-loops from pointed unary theories and section 4 demonstrates how linked-lists and their interface are obtained from the theory of monoids. Finally, section 5 concludes by providing a tabulation of a number of theories and their corresponding data structures.

Our exposition will be informal for brevity and accessibility, however the fact that the proof obligations *forced* the common interfaces to arise was due to exploration in mechanising theories in the proof assistant Agda, which did not allow us to elide any "trivial details". The full details of the theories mentioned in section 5 can be found at: https://github.com/JacquesCarette/TheoriesAndDataStructures

2 COMPOSITIONALITY SCHEMES

'Stepwise refinement' is the idea that programming beings with the empty, or do-nothing, program `skip` then new programs `S ⨾ T` are formed by sequencing other programs together. Moreover, sequencing is associative —that is, `(S ⨾ T) ⨾ R = S ⨾ (T ⨾ R)`— and so languages, such as C or Python, do not require parentheses. This is an instance of a *monoid*, which is a structure that consists of a collection called the `Carrier` and operation `_⨾_ : Carrier → Carrier → Carrier` which is associative and has a no-op identity `Id : Carrier`.

2020. Manuscript submitted to ACM
DOI:

In strongly typed languages we cannot just sequence arbitrary programs as we would in, say, Python, only to obtain an incompatibility error at runtime. Instead, we would like to be able to discuss sequencing 'well-typed' programs. This naturally gives rise to *categories*, which consist of a collection called `Objects` —the 'types' of the language— and for each objects `A, B` a collection `A ⟶ B` of " morphisms, programs, from `A` to `B` " —sometimes also denoted `Hom A B`— such that compatible morphisms can be composed to yield new morphisms by means of an operation `_⨾_ : ∀{A B C} → (A ⟶ B) → (B ⟶ C) → (A ⟶ C)` which is associative and has a no-op identity `Id : ∀{A} → A ⟶ A`. We are using a long arrow '⟶' to refer to morphisms, programs, of a category and a short arrow '→' to refer to usual function typing.

Slogan 0: Monoids model untyped programming, categories model typed programming.

A common task for a programmer is to make inferences about large data sets. One begins with an empty database then aggregates data —here's a monoid! Does it matter *when* inferences are made? If we have no data, then we should be able to make no new, useful, inference. If we have we have two data-sets x and y, we may aggregate them to produce a new data-set `x ⨾ y` from which we may make an inference **I**`(x ⨾ y)`. If we have multiple people working together, it may be ideal to make inferences **I** `x` and **I** `y` in 'parallel' then combine the inferences together to obtain a new inference **I** `x ⨾'` **I** `y`. Of-course, we would like these different approaches to yield the same results, which they may not. In the case they do, we say **I** `: DataSets → Inferences` is a *(homo)morphism.*

As '(homo)morphism', the name of structure-preserving operations between monoids, suggests, we have a category whose objects consist of monoids! Formally, a morphism of monoids **I** `: M ⟶ N` is a function `Carrier M → Carrier N` that sends `Id M` to `Id N` and distributes over composition in `M` to obtain composition in `N`.

The typed analogue of homomorphism is called a 'functor'. A functor `F = (F₀, F₁) :  ⟶` $\mathcal{D}$ between categories is a homomorphism —in that it must preserve the identities and the compositional structure— and it must also preserve the typing: If `f : A ⟶ B` in $\mathcal{C}$, then `F₁ f : F₀ A ⟶ F₀ B` in $\mathcal{D}$. For example, if were Haskell and $\mathcal{C}$ where C, and $\mathtt{F}_0$ was an assignment of Haskell types to C types, then $\mathtt{F}_1$ must preserve typing in that the type of a transformed program `F₁ f` is obtainable from the type of the original program `f`. It is traditional to drop the subscripts and refer to `F₀, F₁` simply by `F` —contextual inference eliminates any ambiguity.

Since categories have nebulous objects, functors also act as a means to endow objects with structure. For example, pairs of integers are just integers under the "pairs of" operation. More formally, let `Bin₀ A = A × A = { (x, y) | x, y ∈ A }` and `Bin₁ f (x, y) = (f x, f y)`. Since `Bin₁ id (x, y) = (id x, id y) = (x , y) = id (x, y)` and `Bin₁ (f ⨾ g) (x, y) = ( g (f x), g (f y) ) = (Bin₁ f ⨾ Bin g) (x, y)`, one easily finds `Bin = (Bin₀, Bin₁)` to be a functor that forms pairs. The operation of forming pairs in general, for any `A` and `B`,~ is a bifunctor denoted "×", with no subscripts as in `A × B` and `f × g`. Likewise, lists of characters are just characters under the "lists of" operation. We encourage the reader to formalise the lists functor and check that it, and the pairs functor, indeed satisfy the functor laws.

Slogan 1: Homomorphisms are structure-preserving operations, functors are homomorphisms that also preserve typing. Moreover, functors are structure.

Any general programming language worth its name would support a form of polymorphism —the ability to declare a cookie-cutter recipe applicable to a family of different types. Monoids no longer provide for such an abstraction —they have no notion of types, how could they! Consider the doubling functions:

```
dupInt : Int → Int × Int
dupInt x = (x , x)
```

```
dupChar : Char → Char × Char
dupChar c = (c, c)
```

We are forced to repeat the definition for each type we are interested in. Polymorphic functions remedy this shortcoming by allowing a *write once, use many* approach:

```
{- A natural transformation from the identity functor to the pairing functor -}
dup : ∀{A} → A → A × A
dup a = (a , a)
```

When such a function makes *no dependence on A*, we say that it is *parametric polymorphic* and that it comes with an optimisation law known as 'naturality': For any operation `f` we have `(f × f) ∘ dup = dup ∘ f`. Less cryptically, this says that expressions of the form `let (x, y) = dup a; (x', y') = (f x, f y) in ⋯` may be replaced with the `let a' = f a; (x' , y') = dup a' in ⋯`. However, polymorphism does not always come in this form; for example, languages such as `C#` which allow type inspection would allow us to form the following polymorphic method which is not natural:

```
first : ∀{A} → A × A → A
first {A} (x, y) = if A is Int then 0 else x
```

A *natural transformation* is a family η `: ∀{A} : F A ⟶ G A` of morphisms such that for any $f : A \longrightarrow B$ we have $Ff \mathbin{⨾} \eta_B = \eta_A \mathbin{⨾} Gf$. If one thinks of `F` and `G` as structures or formats, then η is tantamount to uniform restructuring. Less cryptically, the constraint says that the ways to 'rename', 'relabel', 'transform' `F A` to `G B` using any `f`, are identical: We may rename by operating over the F-structure then reorganise using η, or reorganise using η first then rename by operating over the resulting G-structure.

The reader is encouraged at this point to find an `f : A → B` showing that `first` above fails to be a natural transformation.

Slogan 2: Functors are structures and natural transformations are uniform restructuring schemes.

A program `A ⟶ B` can be thought of as the careful manipulation of `A`-data to yield `B`-data. When `A` and `B` are different representations of the same data, the operation is invertible. Just as categories model typed programming languages, the notion of non-lossy protocols is modelled by the concept of *isomorphism.* An isomorphism, denoted `A ≅ B`, is a pair of morphisms `f : A ⟶ B` and `g : B ⟶ A` that "undo" each other: `f ⨾ g = Id = g ⨾ f`. More concretely, this condition becomes:

$$\forall a, b \bullet \qquad f\ a = b \quad \equiv \quad a = g\ b$$

More often than nought, programs `A ⟶ B` are not invertible but do have a *best approximate inverse.* One writes $f \dashv g$ to indicate this relationship. When there are notions of 'approximation', denoted '≤', the constraint becomes:

$$\forall a, b \bullet \qquad f\ a \le_B b \quad \equiv \quad a \le_A g\ b$$

Strict equalties have been replaced with approximations instead.

For example, the injection $\mathbb{Z}$ and the dup-lication function from earlier have no inverse. However, they do have best approximate inverses:

$$\lceil r \rceil \le_{\mathbb{Z}} n \quad \equiv \quad r \le_n$$

$$x \uparrow y \le_z \quad \equiv \quad (x, y) \le_\times \mathrm{dup}\, z$$

$$p \wedge q \Rightarrow r \quad \equiv \quad (p \Rightarrow r) \wedge (q \Rightarrow r) \text{ i.e., } (p, q) \Rightarrow {}_{\mathbb{B}\times\mathbb{B}}\mathrm{dup}\, r$$

- ◇ The ceiling $\lceil r \rceil$ of a number is the largest *whole* number that is approximated by r:
  1. It is a whole number, $\lceil \_ \rceil :\to \mathbb{Z}$.
  2. It is approximated by r: Taking $n \coloneqq \lceil r \rceil$ in the characterisation yields $r \leq \lceil r \rceil$.
  3. If r approximates another number, say n, then $\lceil r \rceil$ approximates it too! This is just the ‘ $\Leftarrow$ ’ reading of the characterisation.
- ◇ The maximum x↑y is the largest *single* number that is approximated by both x and y.
- ◇ The conjunction $p \wedge q$ is the largest *single* Boolean approximating both Booleans p and q.

Generalising on the duplication example, the reader is encouraged to verify sup $\dashv K$, where the constant function $K :\to (\to)$ takes an element z to the function $(Kz)x = z$ and sup $: (\to) \to$ takes a function f to its supremum sup f.

Slogan 3: For familiar or simple g, one can find useful or complex f with `f ⊣ g`.

For two types `A` and `B`, there may be a number of ways that one “approximates” the other. In a category, we may simply say any morphism `f : A ⟶ B` witnesses such an approximation. With this in-hand, the previous formulation lifts to the categorical setting as follows. For functors `L :` $\mathcal{C} \longrightarrow \mathcal{D}$ `: R`, one says that L *is adjoint to* R, denoted `L ⊣ R`, when there is an isomorphism, as follows, natural in `A, B`.

$$\forall A, B \bullet \qquad L\,A \longrightarrow_{\mathcal{D}} B \quad \equiv \quad A \leq_{\mathcal{C}} R\,B$$

This formulation is terse and easily motivated from the simpler setting, however for verification purposes it is a bit difficult to work with. There is a more ‘local’ formulation.

An *adjunction* $L \dashv R$ consists of two (not necessarily natural!) transformations $\eta : Id \to RL$ and $\epsilon : LR \to Id$ such that

$$\forall f, g \bullet \qquad f = \eta \mathbin{⨾} R\,g \quad \equiv \quad L\,f \mathbin{⨾} \epsilon = g$$

Recall that we may construe functors F as structure, then maps $X \to FX$ provide ways to produce structured elements and so are referred to as *F-algebras./* In particular, transformations $X \to FX$ may be thought of as injecting or “boxing up” elements with a trivial F-structure whereas transformations $FX \to X$ can be thought of as “compiling down” the structure to obtain a concrete value.

With such a terminology, the above characterisation reads: *Each L-algebra g is uniquely determined —as an L-map followed by an ϵ-reduction— by its restriction to the unit η.* Later in the setting of monoids and lists, this becomes: List homomorphisms are uniquely determined, as a map followed by a reduce, by their restriction to the singleton lists.

It can be shown that the transformations are actually natural. As such, a more local formulation of $L \dashv R$ consists of a pair of natural transformations $\eta : Id \to RL$ and $\epsilon : LR \to Id$ such that the “zig-zag” laws holds: $Id = \eta \mathbin{⨾} R\epsilon$ and $Id = L\eta \mathbin{⨾} \epsilon$. This is the formulation we shall follow in the remainder of the exposition.

The remainder of the exposition moves slogan 3 from primitive types to the more complex types that programmers are generally interested in. We begin with simple theories, form the forgetful functor R, then seek to find the free functor L. In the process of establishing the adjunction $L \dashv R$ we are forced to construct the following tool-kit:

Type Constructor: We have a type constructor L that furnishes raw types with structure.

Map: Functions $A \to B$ can be lifted to work on L-structures yielding maps $L\,A \to L\,B$. In database/C# settings, this is known as `select`. Moreover, this operation is a homomorphism. In Haskell notation,
- ◇ `map id = id`
- ◇ `map (f ⨾ g) = map f ⨾ map g`

Wrap: We obtain a way η to construe raw data as having ‘singleton’ structure.

Interpreter: We obtain a way ϵ to (recursively) "fold" over a structured value to obtain a single value.
Moreover, the zig-zag laws ensure that forming a singleton then reducing it is a no-op, as expected.

3 DYNAMICALSYSTEMS —POINTED UNARY THEORIES

A *pointed unary theory* consists of a type, a default value of that type, and an operation on that type. Think of a box with a screen displaying the current state and a button that alters the state. Since such basic computing automata are an instance of such a theory and there is no standard name for the theory's operation, we shall refer to the operation as the "next operation" since it provides a next value in the type. Moreover, we may also refer to these theories as "dynamical systems" since they mimic automata.

Nearly any useful data-structure is an instance of this interface. As such, two simple examples more than suffice.

- The naturals numbers $\mathbb{N}$ with starting state 0 and next operation being the successor function.
- The automata with state space `{even, odd}`, starting state `even`, and next operation `even ↦ odd ↦ even`. Consequently, the induced finite-state machine, `foldl (const next) start xs`, informs us whether a string input `xs` has even or odd length.

If we wish to take dynamical systems to be the objects of a category, call it $\mathcal{DS}$, we must form a notion of homomorphism. The obvious thing to do is to say a homomorphism $h : X \longrightarrow Y$ is a function between the state spaces that preserves the point —i.e., it sends the default point of X to the default point of Y— and it commutes with the 'next' operation: $\forall x \bullet\ h(\mathrm{next}_X\, x) = \mathrm{next}_Y(h\, x)$. It is then a simple exercise to show that the identity function is a homomorphism and the functional composition of homomorphisms results in a homomorphism.

Let $\mathcal{R} : \mathcal{DS} \to \mathcal{S}et$ be the function that yields the underlying state space of a dynamical system. In particular, on objects it 'forgets' the default point and the next operation, simply yielding a set. On homomorphisms, it forgets the structural-preservation proofs and simply yields a function on sets. This is our "forgetful" functor.

How do we form a "free functor" $\mathcal{L} : \mathcal{S}et \to \mathcal{DS}$? We could serialise programs over dynamical systems by simply keeping track of the constructors for the default element and the next operation. Then we could interpret, execute, or run such a program later provided we have a way dyanmical system in hand. Whence, we consider forming *terms* over dynamical systems:

```
{- Dynamic system terms over a variable set A. -}
data Term (A : Set) where
  {- variables are terms -}
  inject  : A → Term A
  {- Function “names” applied to terms are again terms -}
  default : Term A
  next    : Term A → Term A
```

Let's provide a more informative renaming:

```
data Possibly (A : Set) where
  never  : Possibly A
  now    : A → Possibly A
  later  : Possibly A → Possibly A
```

Well that is definitely interesting; it seems we have stumbled upon a modal-like data-structure. A value of type `Possibly A` may `never` be obtained, or it can be obtained `now x` or it is deferred to a later time `later p`. By traversing such abstract syntax trees and altering elements as one sees them, we obtain a `map` operation that makes this type into a functor, call it $\mathcal{L}$. Moreover, for each type `A` this functor yields a dynamical system `(Possibly A, never, later)`.

We are nearly done with our analysis of pointed unary theories. To show that $\mathcal{L} \dashv \mathcal{R}$, we need embedding and evaluation polymorphic functions.

```
η : ∀ {D} → D → Possibly (States D)  {- ≈  Id D → ℒ (ℛ D)  -}
η = now

ϵ : ∀ {D} → Possibly D → States D    {- ≈ ℛ (ℒ D) → Id D -}
ϵ {D} never      =  default D
ϵ {D} (now d)    =  d
ϵ {D} (later pd) =  next D (ϵ pd)
```

Since no type inspection is performed, there are easily shown to be natural transformation. The zig-zag laws are equally trivial.

The type `Possibly A` consists of dynamical system terms *over* the 'variable set' `A`. What if we considered closed terms; i.e., omitting variables altogether. Then the injection `now` can never be invoked and so may be removed; along with renaming the other constructors we obtain:

```
data 𝒩 where
  zero : 𝒩
  succ : 𝒩 → 𝒩
```

It seems that the natural numbers were not just an instance of dynamical systems but rather were canonically so: This type contains the minimum to be considered a dynamical system! There is the carrier state space , the default `zero`, and the next operation `succ`. Moreover, its relationship to other dynamical systems is that it is "initial": There is a homomorphism from it to any other dynamical system, as follows.

```
{- Essentially: succ zero ↦ next(default D) -}
for-loop : ∀ {D} → 𝒩 → States D
for-loop {D} zero     =  default D
for-loop {D} (succ n) =  next D (for-loop)
```

Perhaps the last thing we would have expected would have been for the humble C-style for-loop to appear.

In summary, by looking at the free structures of pointed unary theories we have obtained:

(1) the `Possibly` data type, which is functorial;
(2) ways to serialise programs over dynamical systems and evaluate them;
(3) the natural numbers;
(4) the for-loop.

4 LISTS —MONOIDAL THEORIES

Unsurprising monoids along with monoid homomorphisms also form a category, call it $\mathcal{Mon}$.

Let $\mathcal{R} : \mathcal{Mon} \longrightarrow \mathcal{Set}$ be the functor that forgets the structure —the composition operator and the no-op element— to yield a set. On morphisms, it forgets the proofs to yield a function on sets. This is our forgetful functor.

As we mentioned earlier, to find the free structure associated with this theory we turn to terms over the theory:

```
{- Monoidal terms over ‘variables’ A -}
data Term (A : Set) where
  {- Variables are terms -}
  inj  : A → Term A
  {- Function “names” applied to terms are terms. -}
  Id   : Term A
  _⨾_  : Term A → Term A → Term A
```

The pointed unary setting was lawless, whereas monoids have laws stating how the pieces interact: Composition is associative with unit the no-op. In particular, `inj x ⨾ Id = inj x` *should* hold but it does not. This issue is the lack of canonicity: There are multiple forms for items that should be identical.

If we view the monoid laws as rewrite rules, then it suffices to consider a type of only normal forms and have arbitrary terms rewrite down to them.

```
 Id ⨾ t      ↦  t
 s  ; Id     ↦  s
 (r ⨾ s) ⨾ t  ↦  r ⨾ (s ⨾ t)
```

With these rewrites, an arbitrary term reduces to the form `inj x0 ⨾ (inj x1 ⨾ (inj x2 ⨾ (⋯ ⨾ inj xN)))`; this immediately suggests the so-called “cons lists”: We force this right-parenthesising by having the left be a raw element and the right be a complex element.

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

Had we use the associativity rule as a rewrite rule the other way around, we would have obtained “snoc lists”, which are lists with constant time access to the last element. Hence there are multiple *presentations* of canonical terms, it is enough to pick one and continue with that. The composition operator is regained by rewriting down into the canonical form —by discarding units and parenthesising right-wards:

```
_++_ : ∀ {A} → List A → List B
[] ++ ys        = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

Walking along the pointers of a linked-list and altering values as we see them provides the `map` operation, which is also known as `foreach` loop in Java and imperative languages. It is easily seen to be functorial and so we have a functor, call it $\mathcal{L} : \mathcal{S}et \longrightarrow \mathcal{M}on$.

In order to show that $\mathcal{L} \dashv \mathcal{R}$, we need to formulate embedding and evaluation polymorphic functions.

```
wrap : ∀ {M} → Carrier M → List (Carrier M)
wrap m = m :: []

fold : ∀ {M} → List (Carrier M) → Carrier M
fold {M} []        = Id M
fold {M} (x :: xs) = x ⨾ fold xs where _⨾_ = _⨾_ M
```

Notice that there is only one closed canonical term: If we take the variable set `A` to be empty, we can never invoke the `_::_` constructor and so only have the one value `[]`. If we instead consider the free structure over a *singleton* set, taking `A` to have one irrelevant value, and renaming, yields the naturals again:

```
{- 𝒩 ≅ List ⊤ -}
data 𝒩 : Set where
  zero : 𝒩
  succ : 𝒩 → 𝒩
```

In summary, by considering the free structure associated with the most ubiquitous form of composition we obtained:

(1) The linked-list data-structures, forwards with cons and backwards with snoc;
(2) The `map`, or “foreach”, looping construct; along with its optimisation laws: `map id = id` and `map (f ⨾ g) = map f ⨾ map g`;
(3) The helpful `wrap` function that embeds a type into the assocaited type of lists;
(4) The fold recursion scheme, which is essentially looping.

(5) The adjunction property is tantamount to: List homomorphisms are uniquely determined, as a map followed by a reduce, by their restriction to the singleton lists.

All this from the tiny theory of monoids!

5 DATASTRUCTURES ⊣ THEORIES

Our repository contains many worked out details of how simple theories give rise to interesting or common data-structures. The previous two sections demonstrated the general process, when possible, and there is little to be gained by such repetition. Instead we shall settle for a listing of results followed by remarks about some theories that gave us unexpected trouble.

Two Sorted: A two sorted theory consists of just two types and nothing more.

There are two forgetful functors, depending on which sort is kept. The free structure is then to keep the current sort and declare the new other sort to be empty.

Interestingly, another way to 'forget' the two sorts is to produce the Cartesian product, which is a single type. This gives rise to the pairing "×" functor, whose left adjoint is then the duplication functor —c.f., `dup` from earlier. Surprisingly, duplication, denote it by , itself has left-adjoint: If we think of `A` as 'forgetting' we had a single type and instead thinking we have two types, then given any two types, the free single type is obtained from their disjoint union.

Whence, ⊣ ⊣ ×.

Notable programming combinators:

- ⋄ Records from products;
- ⋄ Projections and structural maps over products;
- ⋄ Enumerations from disjoint sums;
- ⋄ injections and structural maps over sums;
- ⋄ duplication combinators and optimisation laws from naturality conditions.

Relations: A *heterogenous relation* is essentially a binary predicate.

There are at least two forgetful functors to $\mathcal{S}et$, depending on whether we keep the source or the target of the relation. The free structures are, surprisingly, the empty relations.

Upon further reflection, this is rather reasonable. A relation is essentially a graph and if we are given a set of vertices, then the smallest graph that contains such a set must be the empty graph on that vertex set.

Pointed: A pointed theory consists of a type along with a single elected point.

These model types with default values, as in the case in C#.

The forgetful functor is obtained by dropping the point. The free structure is obtained by adjoining a type with a new formal element, sometimes called `null`.

Notable programming combinators:

- ⋄ Nullable types;
- ⋄ the Maybe monad

Unary: A unary theory is a type along with a single unary function on it.

Dropping the function gives us a forgetful functor, whereas the free structure gives us a modal-like data-structure:

```
data Eventually (A : Set) where
  now   : A → Eventually A
  later : Eventually A → Eventually A
```

This type appears silently in the form of `A` values tagged by natural numbers, since `Eventually A ≅ A × ℕ`. It gives us a structure for indicating "how many (delayed) steps" were needed before we obtained a value.

Notable programming combinators:

- ⋄ A novel modal-like type;
- ⋄ The evaluator is iteration, `later (now a) ↦ f a`, along with an array of useful utility properties required for the proof obligations.

Involutive: An involutive theory consists of a type along with a unary function `f` on that type such `f ∘ f = id`.

Keeping only the type gives us a forgetful functor. The free structure is obtained by tagging elements with Booleans —the involution then becomes negating the Boolean tag.

Interestingly, there are *two adjunction proofs* corresponding to whether we embed elements by tagging them with 'true' or with 'false'.

Notable programming combinators:

- ⋄ The Booleans;
- ⋄ Boolean negation to swap the tag;
- ⋄ `map` that works on the elements, regardless of the tag.

Indexed Unary: An indexed unary theory consists of a sort along with an indexed family of operations on it: There is a type `Carrier` and an indexing type `I` and a family of "actions" `Op : {i : I} → Carrier → Carrier`.

These model weak forms of automata.

Keeping only the carrier set yields a forgetful functor. The free structure on `A` is obtained by using `NonEmptyLists A` as the carrier and `A` as the index set, with list concatenation as the family of operators: For each `a : A`, we have an action `a ::_`.

Notable programming combinators:

- ⋄ Fold is a homomorphism from lists from the index set to an indexed unary theory, over the same index set.
- ⋄ Non-empty lists.

Magma : A magma is just a sort along with a binary operation.

Dropping the operation yields a forgetful functor. The data type of binary trees provides a free structure.

Notable programming combinators:

- ⋄ Binary tree data structure;
- ⋄ Recursion schemes over binary trees;
- ⋄ Many coherency laws on how the recursion schemes interact with one another.

N-ary: For given natural number N, an N-ary theory consists of a type along with an endo-operation of N arguments.

Dropping the operation yields a forgetful functor, whereas a free structure is obtained by rose trees.

Semigroup: A semigroup consists of a sort and an associative binary operation.

Keeping only the carrier sort yields a forgetful operation, the type of non-empty lists provides a free structure.

Notable programming combinators:

- ◇ Non empty lists data structure;
- ◇ Catenation of non-empty lists, along with associativity proof;
- ◇ Recursion schemes;

Monoid: A monoid consists is a semigroup with a point that acts as the operation's unit. Keeping only the carrier yields a forgetful operation, whereas linked-lists provide a free structure.

Bag: This is a monoid with the additional law that compositional order does not matter.

Keeping only the carrier yields a forgetful operation, but there is no free structure in a constructive setting.

Viewing the bag axioms as rewrite rules does not yield canonical forms; in particular the commutativity axioms provides a rule `l ⨾ r ↦ r ⨾ l` that can be applied infinitely often.

In classical, non-constructive settings, bags and other theories admit free constructions: One simply forms the terms over the theory than quotients by the equivalence relation induced from the axioms. However, in computing, we want to be able to actually manipulate particular data-values rather than consider nebulous equivalence classes. It seems the leap is not that large, with *decidable equality* in hand, we can form a free structure for bags —but we're no longer in $\mathcal{S}et$ and so no longer in the traditional domain of computing.