Theories and Data Structures

—Draft—

MUSA AL-HASSY, JACQUES CARETTE, WOLFRAM KAHL

1 THEORIES YIELD DATA-STRUCTURES

Since our readership is intended to be a computing audience, there is little to be gained by starting out with data-structures then observing they have a particular algebraic structure. Instead, we propose to begin with tremendously simple interfaces which may arise in computing situations, then briefly form outline their theory. From there, we may then obtain a free theory and discuss its interpretation as a familiar data-structure.

Let us begin with a situation that one encounters almost immediately when learning programming: What default value should you use when writing a loop to aggregate the values in some structure? For example, when multiplying or adding values in a list one may use 1 or 0, respectively, as an initial value for the aggregation —which then acts as the default value for empty lists. Likewise, we may wish to implement a "safe head" operation: Given a list of numbers, we return the first number if it is non-empty and otherwise we return 0 or 1? Hard-coding the default return value forces users to form superfluous conditional expressions, as such it may be better to request a default value to begin with. This gives rise to the following program.

```
{- Types that have a 'default' value -}
record Pointed : Set₁ where
  field
    Carrier : Set
    Point   : Carrier

open Pointed

safe-head : {P : Pointed} → List (Carrier P) → Carrier P
safe-head {P} []       = Point P
safe-head {P} (x :: xs) = x
```

Now that we have demonstrated the utility of the interface, let us analyse its theory.

A "pointed theory" consists of a type along with an elected value of that type; there are no further constraints. A homomorphism, or structure-preserving function, `P` $\longrightarrow$ `Q` of pointed theories is a function between the carries that sends the point of `P` to the point of `Q`. For example, lists over any type form a pointed theory with the point being the empty list, and so the above `safe-head`, by definition, is actually a homomorphism. It is instructive to verify that the identity function is a homomorphism and that the composition of homomorphisms is again a homomorphism. Consequently, pointed theories form a category that we shall call $\mathcal{PS}et$.

If we drop the point from a pointed theory, we are left with a type and so have a 'forgetful functor', say, $\mathcal{R} : \mathcal{PS}et \longrightarrow \mathcal{S}et$. As per our slogans, let us search for a functor $\mathcal{L}$ that is left adjoint to $\mathcal{R}$. Rather than guess an $\mathcal{L}$, we shall

2020. Manuscript submitted to ACM
DOI:

attempt to calculate it by using the adjunction characterisation. Indeed, for any X, Y, y, we aim to find an $\mathcal{L}$ such that $\mathcal{L}X \longrightarrow (Y, y) \quad \cong \quad X \to \mathcal{R}(Y, y)$:

$$\begin{array}{lll}
\text{—In category } \mathcal{S}et & & f \in X \to \mathcal{R}(Y, y) \\
& \equiv & \{ \text{ Definition of } \mathcal{R} \ \} \\
& & f \ \in \ X \to Y \\
& \cong & \{ \text{ Wish we could transform } f \mapsto f', X \mapsto X' \text{ and have } x' : X' \ \} \\
& & f' \ \in \ X' \to Y \quad \wedge \quad f' \, x' = y \\
& \equiv & \{ \text{ Definition of homomorphism } \} \\
& & f' \ \in \ (X', x') \longrightarrow (Y, y) \\
& \equiv & \{ \ \textbf{Define } \mathcal{L}X = (X', x') \ \} \\
\text{—In category } \mathcal{PS}et & & f' \ \in \ \mathcal{L}X \longrightarrow (Y, y)
\end{array}$$

To completely define $\mathcal{L}$, it seems we must associate to any set X a new non-empty set X' such that functions $X \to Y$ correspond exactly to homomorphisms $(X', x') \longrightarrow (Y, y)$. The simplest thing to do is to adjoin X with a new formal element, then f' is defined by sending the new element to y and otherwise acts like f. Conversely, given such an f' we regain f by simply embedding the existing elements of X into X'. We may code up X', f' as `Maybe X`, `⟦ f ⟧` below, and the inverse operation as `lower f`.

```
data Maybe (X : Set) : Set where
  Just    : X → Maybe X    {- Keeping existing elements of X -}
  Nothing : Maybe X       {- Adjoining a new element -}

⟦_⟧ : {X : Set} {Y : Pointed} (f : X → Carrier Y) → Maybe X → Carrier Y
⟦ f ⟧ Nothing  = Point Y  {- The new forced equation for the new element -}
⟦ f ⟧ (Just x) = f x      {- Leaving f to behave on existing elements    -}

lower : {X : Set} {Y : Pointed} (f' : Maybe X ⟶ Y)  →  X → Y
lower f' x = f' (Just x)  {- Embed existing elements of X into X' -}
```

Clearly these operations are inverses and being parametric polymorphic, they are natural transformations. However, for $\mathcal{L}$ to be a functor we also need to define its behaviour on functions, which we do below via `map`, and so we have the necessary pieces for the adjunction. In verifying the full details of $\mathcal{L} \dashv \mathcal{R}$, we find that *every pointed homomorphism is determined uniquely as* `map` *followed by an interpretation* ⟦⟧*!*

```
map : {A B : Set} → (A → B) → (Maybe A → Maybe B)
map f Nothing  = Nothing
map f (Just a) = Just (f a)
```

It is at this stage that we may enumerate the benefits of exploring the simple theory of pointed structures. The goal of finding an adjoint for discarding the pointed structure has resulting in the following useful toolkit.

(1) The `Maybe` data-structure: When one is unsure what would be a useful default value to return, they may simply return `Nothing`.

This provides a standard mechanism to handle unexpected situations: Rather than crash the system or request a default value, yield a formal value and let the user handle it. In particular, object-oriented languages users, such as of C# and Java, do this all the time by returning `null` as a default value.

(2) We have elected to use the name `Maybe`; however this data-structure arises under a number of different names, according to intended usage. Names include:
   - ‘Nullable’: In object-oriented settings, it denotes a value that is completely ‘undefined’ but in a first-class fashion.
   - ‘Optional’: In an method argument location, it denotes arguments that need not be present.

(3) The ‘interpretation’ operation $[\![\,]\!]$ is the method to coerce the formal element into a legitimate element of another data-type.

(4) We have a traversal function, `map`, for working over such structures.

   It comes with two immediate optimisation rules: `map id = id` and `map (f ∘ g) = map f ∘ map g`.

The useful observation and primary contribution of our work is that a data-structure *along* with a minimal infrastructure “falls out” of the adjunction details for discarding additional complexity.