Algebra of Programming **OPLSS 2022**

Lecture 1: Algebraic Fundamentals

Lecturer: Jeremy Gibbons Scribes: Ke Du, Jessica Li, Slim Lim, Cody Rivera, Michelle Thalakottur

1 Introduction

The **algebra of programming** broadly refers to the notion that *the structure of the data dictates the structure of the program.* Several books have explored this correspondence in depth, including *Structured Programming* by Hoare et al.; *The Algebra of Programming* by Bird and de Moor; and *How to Design Programs* by Felleisen et al.

These lectures will cover the following topics:

- Folds and unfolds (catamorphisms and anamorphisms), which establish a basis for structured recursion
- Generalizations of folds and unfolds (paramorphisms, histomorphisms, etc.)
- Metamorphisms, a new research area by Gibbons that provides useful abstractions for streaming and data compression algorithms
- Effectful traversals of data structures.

Much of the literature on the algebra of programming takes a category-theoretic approach, but we will sidestep this overhead by primarily expressing our ideas in Haskell, a lazy functional programming language. Moreover, we restrict our mental framework to describing total functions between sets, sidestepping the complexity of cpos and nontermination.

In this lecture, we introduce the algebraic primitives necessary for our discussion.

2 Products

If A and B are sets, then we can think of $A \times B$ as their **product**:

$$A \times B = \{(a, b) \mid a \in A, b \in B\}$$

The Cartesian product is a particular *explicit construction* of a product, but the essence of the algebra of programming is identifying useful abstractions that generalize meaningfully across different types. For this reason, we prefer to deal with the implicit construction given by some *universal property*: a particular set of requirements that yield a unique solution.

Remark 1 (Universal property)**.** *A universal property defines objects uniquely up to isomorphism.*

In other words, two solutions that satisfy the same universal property are isomorphic. We can therefore define concepts entirely by specifying a universal property.

Remark 2 (Structure of a product)**.** *A product $A \times B$ is a triple $(X, \mathsf{fst}, \mathsf{snd})$ consisting of:*

- *a set X,*
- *functions $\mathsf{fst} : X \to A$ and $\mathsf{snd} : X \to B$*

which together satisfy the universal property, which we will discuss in a moment.

So far, we have only established the shape of a valid solution, not anything about how X, fst, and snd should behave. There are many possible solutions that satisfy Remark 2, but we are interested in finding an *extremal* solution which is "least" or "greatest" in some way.

Intuitively, an extremal solution $(X, \mathsf{fst}, \mathsf{snd})$ is more expressive than any competing solution $(X', \mathsf{fst}', \mathsf{snd}')$. We can formalize this notion of expressivity through the universal property:

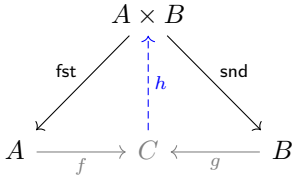
Definition 1 (Universal Property for Products)**.** *Consider two products satisfying the structure defined in Remark 2:*

$$\begin{array}{rlll} (X, \mathsf{fst}, \mathsf{snd}) & X \text{ is a set} & \mathsf{fst} : X \to A & \mathsf{snd} : X \to B \\ (C, f, g) & C \text{ is a set} & f : C \to A & g : C \to B \end{array}$$

We say that $(X, \mathsf{fst}, \mathsf{snd})$ satisfies the universal property if for any choice of (C, f, g) there exists a unique $h : C \to X$ between the two sets, such that the following equations hold:

$$f = \mathsf{fst} \circ h$$
$$g = \mathsf{snd} \circ h$$

We can represent these equations with the following commutative diagram. The dashed arrow for h indicates that h is the unique function between C and $A \times B$.



In other words, if Alice has a triple satisfying the structure in Remark 2, but Bob provides a competing triple, then Alice must somehow prove that her solution is somehow richer than Bob's. She does this by showing there is a unique function h which, precomposed with her solution, produces the same result as Bob's. (In category theory we might say that Bob's solution *factors through* Alice's.)

A triple satisfying the universal property is an extremal solution, and these solutions are unique up to ismorphism. Going forward, we will refer to this unique solution as "the" product, and write it as

$$(A \times B, \mathsf{fst}, \mathsf{snd}).$$

Definition 2 (Categorical Product)**.** *Consider the product $(X, \mathsf{fst}, \mathsf{snd})$, where*

$$X = A \times B, \qquad \mathsf{fst} = \pi_1, \qquad \mathsf{snd} = \pi_2.$$

Then $(A \times B, \mathsf{fst}, \mathsf{snd})$ is an extremal solution, given by the function

$$h(c) = \langle f(c), g(c) \rangle$$

which satisfies the universal property 1 and is unique up to isomorphism.

2.1 Fork

How should we interpret h? Given two generators f and g, the function h "forks" a single input into two results. As a notational convenience, we will define the notation $\triangle$ (pronounced "fork") for h.

Definition 3 (Fork)**.** *Given some $f : C \to A$ and $g : C \to B$, the following notations are equivalent:*

$$h = f \triangle g$$
$$\iff h(c) = \langle f(c), g(c) \rangle$$

We can restate the universal property using $\triangle$ as follows:

$$\begin{aligned} h = f \triangle g &\iff \mathsf{fst} \circ h = f \ \wedge \ \mathsf{snd} \circ h = g \\ &\iff \mathsf{fst} \circ (f \triangle g) = f \ \wedge \ \mathsf{snd} \circ (f \triangle g) = g && \text{substitute } h := f \triangle g \\ &\iff h = (\mathsf{fst} \circ h) \triangle (\mathsf{snd} \circ h) && \text{conversely} \end{aligned}$$

Remark 3. *The definition of $f \triangle g$ creates a pair from two arbitrary generators f and g. Therefore,*

$$\mathsf{fst} \triangle \mathsf{snd} = \mathsf{id}.$$

3 Coproducts

The dual of a product is a **coproduct**, also known as a **sum** or **discriminated union**. An explicit definition of the coproduct of sets A and B might look like this:

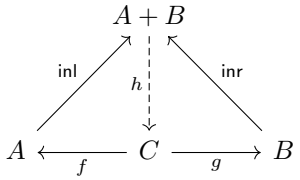
$$A + B = \{(0, a) \mid a \in A\} \cup \{(1, b) \mid b \in B\}$$

where each term is a pair consisting of a *tag* or discriminant describing which set we are drawing from, along with a value from that set.

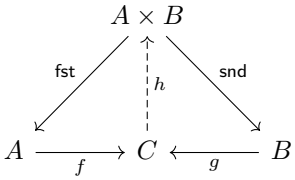
Once again, however, we prefer to define coproducts using the *implicit construction* given by their universal property. We can dualize our construction for products by turning each arrow around, which gives us a new set of functions:

$$\begin{aligned} \mathsf{fst} : A \times B \to A \quad &\text{becomes} \quad \mathsf{inl} : A \to A + B \\ \mathsf{snd} : A \times B \to B \quad &\text{becomes} \quad \mathsf{inr} : B \to A + B \end{aligned}$$

Likewise, we dualize the commutative diagram for products by turning each arrow around:



(a) Coproduct



(b) Product

Dualizing gives us a new extremal solution which satisfies the new universal property:

Definition 4 (Coproduct)**.** *A coproduct is a triple $(X, \mathsf{inl}, \mathsf{inr})$ where X is a set and $\mathsf{inl} : A \to X$ and $\mathsf{inr} : B \to X$ are functions. The extremal solution is defined as follows:*

$$X = A + B, \qquad \mathsf{inl}\, a = (0, a), \qquad \mathsf{inr}\, b = (1, b).$$

The uniqueness of our solution $(A + B, \mathsf{inl}, \mathsf{inr})$ is given by the function

$$h(c) = \mathsf{match}\ c \begin{cases} \mathsf{inl}\, a \mapsto f(a) \\ \mathsf{inr}\, b \mapsto g(b) \end{cases}$$

which satisfies the universal property for coproducts.

3.1 Join

Intuitively, the function h now dispatches on c and applies the correct f or g corresponding to c's tag. In this manner, h performs a **join** on data that could potentially belong to one of two sets.

As with products, we define notation $\triangledown$ (pronounced "join") for h in the coproduct.

Definition 5 (Join)**.** *Given some $f : C \to A$ and $g : C \to B$, the following notations are equivalent:*

$$h = f \triangledown g$$
$$\iff h(c) = \mathsf{match}\ c \begin{cases} \mathsf{inl}\, a \mapsto f(a) \\ \mathsf{inr}\, b \mapsto g(b) \end{cases}$$

We can restate the universal property using $\triangledown$ as follows:

$$h = f \triangledown g \iff h \circ \mathsf{inl} = f \ \wedge \ h \circ \mathsf{inr} = g.$$

4 Functors

The product $\times$ and coproduct $+$ are examples of **functors**, which map between categories. For our purposes, it suffices to think of a functor as a higher-order function:

Definition 6 (Functor)**.** *In the context of this lecture, a functor F defines a mapping:*

- *Between sets $A \mapsto FA$, for all A*
- *Between functions $(f : A \to B) \mapsto (Ff : FA \to FB)$, for all sets A, B*

$$\frac{f : A \to B}{Ff : FA \to FB}$$

Note: In general, we denote functor application without parentheses, but FX and $F(X)$ are equivalent notations.

We are specifically interested in **polynomial functors** on types, which correspond to the algebraic data types used in programming.

Definition 7 (Polynomial functor)**.** *A polynomial functor F can be written using only the following types and operations:*

$$1 \mid X \mid + \mid \times$$

where 1 is the unit type with sole inhabitant $\langle\rangle$, X is the type we are mapping, and $+$ and $\times$ are the product and sum as defined above.

Here are a few examples of polynomial functors, and their corresponding Haskell implementations:

$N(X) = 1 + X$	`data N x = Nothing | Just x`	(Maybe)
$L(X) = 1 + \mathbb{N} \times X$	`data L x = Nil | Cons Nat x`	(List)
$T_A(X) = A + X \times X$	`data T a x = Leaf a | Branch x x`	(Tree)

Each of these functors defines the shape of a type:

Example 1 (Maybe)**.** *$N(X) = 1 + X$ is the shape functor representing the presence or absence of some X.*

- *1 represents the absence of a result.*
- *X represents a result of the given type.*

Example 2 (Linked list cell)**.** *$L(X) = 1 + \mathbb{N} \times X$ is the shape functor for a linked list of natural numbers.*

- *1 represents an empty cell.*
- *$\mathbb{N} \times X$ represents a non-empty cell with some $n \in \mathbb{N}$ consed onto a value of type X.*

Example 3 (Tree node)**.** *$T_A(X) = A + X \times X$ is the shape functor for a binary tree with values of type A.*

- *A represents a value at a leaf.*
- *$X \times X$ represents a branch with a left and right child.*

4.1 Fixed points

The **fixed points** of a polynomial functor F are the types X for which

$$X = FX.$$

In other words, a fixed point is a type that remains unchanged by the shape functor F.

Remark 4. *In general, functors are not guaranteed to have a unique fixed point, or even any fixed points at all. However, all polynomial functors in the category* **Set** *have fixed points.*

As with before, we are interested in the extremal solutions to our fixed point equation. Specifically, we are looking for the **least fixed point** of a given shape functor F.

Definition 8 (Least fixed point)**.** *Let F be a polynomial shape functor in* **Set**.

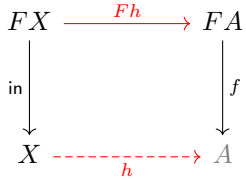
A fixed point of F is a pair $(\mu F, \mathsf{in})$, where $\mathsf{in} : F(\mu F) \to \mu F$ is an isomorphism; that is,

$$\mu F \approx F(\mu F).$$

We say that $(\mu F, \mathsf{in})$ is a least fixed point if, for all other pairs (A, f) with $f : F(A) \to A$, there exists a unique $h : \mu F \to A$ satisfying the universal property

$$h \circ \mathsf{in} = f \circ Fh.$$

Here is the commutative diagram illustrating the universal property for some nonspecific (X, in):



We can set $X = \mu F$ and interpret the diagram as follows:

- F describes a general shape,
- μF is a datatype, the least fixed point of the shape functor F,
- $F(\mu F)$ is a collection of fragments of information for the datatype,
- $\mathsf{in} : F(\mu F) \to \mu F$ takes these fragments of information and turns them into a usable datatype μF.

Here are the aforementioned polynomial functors with descriptions of their least fixed points:

$N(X) = 1 + X$	$\mu N = \mathbb{N}$	(Maybe)
$L(X) = 1 + \mathbb{N} \times X$	$\mu L =$ finite lists of $\mathbb{N}$	(List)
$T_A(X) = A + X \times X$	$\mu T_A =$ finite trees of A	(Tree)

Example 4 (Sum of a list)**.** *Consider the shape functor* $L(X) = 1 + \mathbb{N} \times X$*, and suppose we have the algebra* $(\mathbb{N}, add)$*, with add defined as follows:*

$$\begin{aligned} add &: L\,\mathbb{N} \to \mathbb{N} \\ &: 1 + \mathbb{N} \times \mathbb{N} \to \mathbb{N} \\ add\ (\mathsf{inl}\, x) &= 0 \\ add\ (\mathsf{inr}\ (x, y)) &= x + y \end{aligned}$$

Notice that when we instantiate L with the type $\mathbb{N}$, the case $\mathsf{inr}(x, y)$ is a pair of two natural numbers, rather than a natural number and the tail of a list as we might expect from a function using explicit recursion. Think of *add* as a function being folded over the list, where the non-empty case includes the current element x and an accumulator y representing the sum over elements in the tail.

$$(\underbrace{x}_{\text{head}}, \underbrace{y}_{\text{sum of tail}})$$

The commutative diagram describes two different paths to transform our least fixed point $L(\mu L)$ into a $\mathbb{N}$ via *add*:

$$\begin{array}{lccc} \textit{bits of data structure} & L(\mu L) & \xrightarrow{Lh} & L\,\mathbb{N} \\ & \Big\downarrow{\scriptstyle \mathsf{in}} & & \Big\downarrow{\scriptstyle add} \\ \textit{complete data structure} & \mu L & \xrightarrow[h=sum]{} & \mathbb{N} \end{array}$$

We start with $L(\mu L)$, which corresponds to the pieces of the data structure μL. First, we can use in to transform the pieces of our shape into a μL, then use h, which we call *sum*, to combine the results into a $\mathbb{N}$. Alternatively, we can transform the pieces $L(\mu L)$ into an optional pair of natural numbers, then *add* them together to get the sum of the list.

5 Folds

We will learn more about folds in the next lecture, but we call h the **fold** or `cata` of *add* in the above commutative diagram. More generally, for some least fixed point $(\mu L, \mathsf{in})$ and another algebra (A, f), we define

$$h = \texttt{cata}\ f \quad \iff \quad h \circ \mathsf{in} = f \circ Fh.$$

The Haskell exercise can be found here.