The Algebra and Dissection of Types

Ryan Orendorff[1,2]

May 2016

[1]Department of Bioengineering
University of California, Berkeley
University of California, San Francisco

[2]Innolitics LLC
Medical Imaging Software Consultants

Table of Contents

The Reals

A familiar set of symbols is the reals ($\mathbb{R}$). With the reals we can define an operation $+$ that combines two reals together.

$$+ :: \mathbb{R} \to \mathbb{R} \to \mathbb{R}$$

We note that this function $+$ is *closed* over the set $\mathbb{R}$; applying $+$ to two reals gives something in the same set (another real).

The Reals

A familiar set of symbols is the reals ($\mathbb{R}$). With the reals we can define an operation $+$ that combines two reals together.

$$+ :: \mathbb{R} \to \mathbb{R} \to \mathbb{R}$$

We note that this function $+$ is *closed* over the set $\mathbb{R}$; applying $+$ to two reals gives something in the same set (another real).

We denote this algebra with the following notation for some set $\mathcal{S}$and some operation Op.

$$(\mathcal{S}, Op)$$

Algebraic Structures

Often sets and associated operators have a common thread. For example, for $(\mathbb{R}, +)$ and $(\mathbb{B}, \vee)$

Algebraic Structures

Often sets and associated operators have a common thread. For example, for $(\mathbb{R}, +)$ and $(\mathbb{B}, \vee)$

- The operator return an element in the same set.

$$
\begin{aligned}
+ &:: \mathbb{R} \to \mathbb{R} \to \mathbb{R} \\
\vee &:: \mathbb{B} \to \mathbb{B} \to \mathbb{B}
\end{aligned}
$$

Algebraic Structures

Often sets and associated operators have a common thread. For example, for $(\mathbb{R}, +)$ and $(\mathbb{B}, \vee)$

- The operator return an element in the same set.

$$\begin{aligned} + &:: \mathbb{R} \to \mathbb{R} \to \mathbb{R} \\ \vee &:: \mathbb{B} \to \mathbb{B} \to \mathbb{B} \end{aligned}$$

- The operator is associative.

$$\begin{aligned} a + (b + c) &= (a + b) + c \\ a \vee (b \vee c) &= (a \vee b) \vee c \end{aligned}$$

Algebraic Structures

Often sets and associated operators have a common thread. For example, for $(\mathbb{R}, +)$ and $(\mathbb{B}, \vee)$

- The operator return an element in the same set.

$$
\begin{aligned}
+ &:: \mathbb{R} \to \mathbb{R} \to \mathbb{R} \\
\vee &:: \mathbb{B} \to \mathbb{B} \to \mathbb{B}
\end{aligned}
$$

- The operator is associative.

$$
\begin{aligned}
a + (b + c) &= (a + b) + c \\
a \vee (b \vee c) &= (a \vee b) \vee c
\end{aligned}
$$

- There exists an $e \in \mathcal{S}$ such that $e \times a = a \times e = a$.

$$
\begin{aligned}
a + 0 &= 0 &+ a = a \\
a \vee \mathit{False} &= \mathit{False} &\vee\ a = a
\end{aligned}
$$

These types of algebraic structures are *monoids*.

Table of Contents

Algebraic Data Types

In this talk we are going to focus on only algebraic data types (ADTs)[1]. For example, we can define a type that holds multiple values for one constructor

$$\textbf{data}\ \mathit{Doubled}\ b = \mathit{Doubled}\ b\ b$$

and we can use multiple constructors for a type.

$$\textbf{data}\ \mathit{Cards} = \mathit{Hearts} \mid \mathit{Diamonds} \mid \mathit{Clubs} \mid \mathit{Spades}$$

Let's look at each of these cases in turn.

[1] we are actually going to further restrict to regular data types

Products

We can define a product type as follows, where we hold onto two values.

$$\textbf{data}\ Product\ a\ b = Product\ a\ b$$

Products

We can define a product type as follows, where we hold onto two values.

$$\textbf{data}\ Product\ a\ b = Product\ a\ b$$

Which is equivalent to a pair tuple.

$$Product\ a\ b \cong (a, b) = (,)\ a\ b$$

Products

We can define a product type as follows, where we hold onto two values.

$$\textbf{data}\ Product\ a\ b = Product\ a\ b$$

Which is equivalent to a pair tuple.

$$Product\ a\ b \cong (a, b) = (,)\ a\ b$$

We can write the same thing in a slightly different way.

$$\textbf{data}\ a \times b = a \hat{\times} b$$

The One

Now let's do a little trick. Let's define the following data type.

$$\mathbf{data}\ 1 = \hat{1}$$

We can only construct this value in one way. How many ways are there to construct this?

$$1 \times a$$

[2] $(\cong)$ is usually read as "equal up to isomorphism"

The One

Now let's do a little trick. Let's define the following data type.

$$\mathbf{data}\ 1 = \hat{1}$$

We can only construct this value in one way. How many ways are there to construct this?

$$1 \times a$$

Well 1 only has one value ($\hat{1}$), so we can think of the following relation to hold[2].

$$1 \times a \cong a \cong a \times 1$$

[2] ($\cong$) is usually read as "equal up to isomorphism"

Product is a monoid!

Remember that a monoid has to follow the following rules.

[3] The star $*$ represents a kind.

Product is a monoid!

Remember that a monoid has to follow the following rules.

- The operator return an element in the same set[3].

$$a \times b :: * \rightarrow * \rightarrow *$$

[3]The star $*$ represents a kind.

Product is a monoid!

Remember that a monoid has to follow the following rules.

- The operator return an element in the same set[3].

$$a \times b :: * \to * \to *$$

- The operator is associative.

$$a \times (b \times c) \cong (a \times b) \times c$$

[3] The star $*$ represents a kind.

Product is a monoid!

Remember that a monoid has to follow the following rules.

- The operator return an element in the same set[3].

$$a \times b :: * \to * \to *$$

- The operator is associative.

$$a \times (b \times c) \cong (a \times b) \times c$$

- There exists an $e \in \mathcal{S}$ such that $e \cdot a = a \cdot e = a$.

$$a \times 1 \cong 1 \times a \cong a$$

[3] The star $*$ represents a kind.

Sums

We can define a sum type as follows, where we hold onto either one value or a different value.

$$\textbf{data}\ Sum\ a\ b = Sum\ a\ b$$

Sums

We can define a sum type as follows, where we hold onto either one value or a different value.

$$\mathbf{data}\ Sum\ a\ b = Sum\ a\ b$$

Which is equivalent to this.

$$Sum\ a\ b \cong a \mid b$$

Sums

We can define a sum type as follows, where we hold onto either one value or a different value.

$$\textbf{data}\ Sum\ a\ b = Sum\ a\ b$$

Which is equivalent to this.

$$Sum\ a\ b \cong a \mid b$$

We can write the same thing in a slightly different way.

$$\textbf{data}\ a + b = L\ a \mid R\ b$$

Zero is rather empty

Now let's do a little trick. Let's define the following data type.

$$\mathbf{data}\ 0$$

Zero is rather empty

Now let's do a little trick. Let's define the following data type.

$$\textbf{data}\ 0$$

How do we construct a value of this type? We can't! We could never call the following function.

$$uncallable :: 0 \to a$$
$$uncallable\ x = x\ \text{`}seq\text{`}\ error\ \texttt{":-("}$$

Zero is rather empty

Now let's do a little trick. Let's define the following data type.

$$\textbf{data}\ 0$$

How do we construct a value of this type? We can't! We could never call the following function.

$$uncallable :: 0 \to a$$
$$uncallable\ x = x\ \text{`}seq\text{`}\ error\ \texttt{":-("}$$

In which case, we expect the following.

$$a + 0 \cong 0 + a \cong a$$

Sum is a monoid!

Remember that a monoid has to follow the following rules.

[4] The star $*$ represents a kind.

Sum is a monoid!

Remember that a monoid has to follow the following rules.

- The operator return an element in the same set[4].

$$a + b :: * \to * \to *$$

[4] The star $*$ represents a kind.

Sum is a monoid!

Remember that a monoid has to follow the following rules.

- The operator return an element in the same set[4].

$$a + b :: * \to * \to *$$

- The operator is associative.

$$a + (b + c) \cong (a + b) + c$$

[4] The star $*$ represents a kind.

Sum is a monoid!

Remember that a monoid has to follow the following rules.

- The operator return an element in the same set[4].

$$a + b :: * \to * \to *$$

- The operator is associative.

$$a + (b + c) \cong (a + b) + c$$

- There exists an $e \in \mathcal{S}$ such that $e \cdot a = a \cdot e = a$.

$$a + 0 \cong 0 + a \cong a$$

[4] The star $*$ represents a kind.

Constant and Identity Types

We are also going to define the following data types.

Constant and Identity Types

We are also going to define the following data types.

A constant type, which represents something like Nil or 5.

$$\textbf{data}\ K\ x = \hat{K} \quad \text{-- constant}$$

Constant and Identity Types

We are also going to define the following data types.

A constant type, which represents something like Nil or 5.

$$\textbf{data}\ K\ x = \hat{K} \quad \text{-- constant}$$

An identity type.

$$\textbf{data}\ Id\ x = \hat{Id}\ x \quad \text{-- element}$$

Semirings

A semiring is defined by the following set of rules.

Semirings

A semiring is defined by the following set of rules.

- $(+, 0)$ is a commutative monoid.

Semirings

A semiring is defined by the following set of rules.

- $(+, 0)$ is a commutative monoid.
- $(\times, 1)$ is a monoid.

Semirings

A semiring is defined by the following set of rules.

- $(+, 0)$ is a commutative monoid.
- $(\times, 1)$ is a monoid.
- $\times$ distributes over $+$: $a \times (b + c) = a \times b + a \times c$ and $(b + c) \times a = b \times a + c \times a$.

Semirings

A semiring is defined by the following set of rules.

- $(+, 0)$ is a commutative monoid.
- $(\times, 1)$ is a monoid.
- $\times$ distributes over $+$: $a \times (b + c) = a \times b + a \times c$ and $(b + c) \times a = b \times a + c \times a$.
- 0 is an annihilator for $\times$: $a \times 0 = 0 \times a = 0$.

Semirings

A semiring is defined by the following set of rules.

- $(+, 0)$ is a commutative monoid.
- $(\times, 1)$ is a monoid.
- $\times$ distributes over $+$: $a \times (b + c) = a \times b + a \times c$ and $(b + c) \times a = b \times a + c \times a$.
- 0 is an annihilator for $\times$: $a \times 0 = 0 \times a = 0$.

Semirings

A semiring is defined by the following set of rules.

- $(+, 0)$ is a commutative monoid.
- $(\times, 1)$ is a monoid.
- $\times$ distributes over $+$: $a \times (b + c) = a \times b + a \times c$ and $(b + c) \times a = b \times a + c \times a$.
- 0 is an annihilator for $\times$: $a \times 0 = 0 \times a = 0$.

ADTs form a semiring!

Star (or Closed) Semiring

A star semiring or closed semiring has the additional unary operator $-^*$, defined as

$$a^* = 1 + aa^* = 1 + a^*a$$

[5] This intuition is not correct for all closed semirings

Star (or Closed) Semiring

A star semiring or closed semiring has the additional unary operator $-^*$, defined as

$$a^* = 1 + aa^* = 1 + a^*a$$

Intuitively, $a^* = 1 + a + a^2 + a^3 + \ldots$.[5] What would this look like for types?

[5] This intuition is not correct for all closed semirings

Star (or Closed) Semiring

A star semiring or closed semiring has the additional unary operator $-^*$, defined as

$$a^* = 1 + aa^* = 1 + a^*a$$

Intuitively, $a^* = 1 + a + a^2 + a^3 + \ldots$.[5] What would this look like for types?

For lists, we can have lists of no length or one length or ...

$$\textbf{data}\ List\ a = [\,] \mid a \mid (a, a) \mid (a, a, a) \mid \ldots$$

[5] This intuition is not correct for all closed semirings

Star (or Closed) Semiring

A star semiring or closed semiring has the additional unary operator $-^*$, defined as

$$a^* = 1 + aa^* = 1 + a^*a$$

Intuitively, $a^* = 1 + a + a^2 + a^3 + \ldots$.[5] What would this look like for types?

For lists, we can have lists of no length or one length or ...

$$\textbf{data}\ \mathit{List}\ a = [\,]\ |\ a\ |\ (a, a)\ |\ (a, a, a)\ |\ ...$$

ADTs can be described as a closed semiring.

[5] This intuition is not correct for all closed semirings

Table of Contents

Zippers are great for constant (locally) updating structures

Zippers are a convenient way of traversing a structure, keeping track of where you are.

Zippers are also sometimes known as a "one hole context"

Zippers are great for constant (locally) updating structures

Zippers are a convenient way of traversing a structure, keeping track of where you are.

Zippers are also sometimes known as a "one hole context"

$$\mathbf{data}\ List = [\,]\mid a : (List\ a)$$

$$\mathbf{data}\ Zipper\ a = Zipper\ [a]\ [a]$$

Zippers are great for constant (locally) updating structures

Zippers are a convenient way of traversing a structure, keeping track of where you are.

Zippers are also sometimes known as a "one hole context"

$$\textbf{data}\ List = [\,]\ |\ a : (List\ a)$$

$$\textbf{data}\ Zipper\ a = Zipper\ [\,a\,]\ [\,a\,]$$

Zippers allow for constant time *local* updates to a data structure.

Zippers are great for constant (locally) updating structures

Zippers are a convenient way of traversing a structure, keeping track of where you are.

Zippers are also sometimes known as a “one hole context”

$$\mathbf{data}\ List = [\,]\ |\ a : (List\ a)$$

$$\mathbf{data}\ Zipper\ a = Zipper\ [\,a\,]\ [\,a\,]$$

Zippers allow for constant time *local* updates to a data structure.

Zippers are used in extensively in XMonad.

Table of Contents

Derivatives can also lead to a one hole context

The derivative of a type is often known as a “one hole context”.
We can think of this as marking where we are in a data structure.

Derivative of a Constant

We know from calculus a few derivative rules. The basics are that the derivative of a constant is zero.

$$\partial K = 0$$

Derivative of a Constant

We know from calculus a few derivative rules. The basics are that the derivative of a constant is zero.

$$\partial K = 0$$

How does this look for a value of type K?

$$\partial Nil = 0$$

A value has no context; it is the only thing in town!

Derivative of a variable

The derivative of variable by itself is 1.

$$\partial Id = 1$$

Derivative of a variable

The derivative of variable by itself is 1.

$$\partial Id = 1$$

How does this look for a value of type Id?

$$\partial(\hat{Id}\ x) = ()$$

Derivative of a Product

The derivative of a product is described by the product rule[6].

$$\partial(f \times g) = \partial f \times g + f \times \partial g$$

[6]or more generally by the Leibniz rule

Derivative of a Product

The derivative of a product is described by the product rule[6].

$$\partial(f \times g) = \partial f \times g + f \times \partial g$$

If we take the algebraic form of $x \times x$.

$$\partial(x \times x) = 1 \times x + x \times 1$$

The values can be either $L\,(\hat{1} \hat{\times} x)$ or $R\,(x \hat{\times} \hat{1})$

[6] or more generally by the Leibniz rule

Derivative of a Product

The derivative of a product is described by the product rule[6].

$$\partial(f \times g) = \partial f \times g + f \times \partial g$$

If we take the algebraic form of $x \times x$.

$$\partial(x \times x) = 1 \times x + x \times 1$$

The values can be either $L\,(\hat{1} \hat{\times} x)$ or $R\,(x \hat{\times} \hat{1})$

This represents that we can either be in the left side of a product or the right hand side.

[6] or more generally by the Leibniz rule

Derivative of a Sum

The derivative of a sum is defined by the following rule.

$$\partial(f + g) = \partial f + \partial g$$

Derivative of a Sum

The derivative of a sum is defined by the following rule.

$$\partial(f + g) = \partial f + \partial g$$

If we take the algebraic form of $x + 1$.

$$\partial(x + 1) = 1 + 0$$

Table of Contents

Dissection is a generalization of derivatives

When we calculate the derivative of a type, we don't differentiate the values to the left of the hole from the values to the right.

Dissection is a generalization of derivatives

When we calculate the derivative of a type, we don't differentiate the values to the left of the hole from the values to the right.

Take for example a $List$.

$$\partial List\ a = (List\ a, List\ a) \quad \text{-- the same as Zipper}$$

Dissection is a generalization of derivatives

When we calculate the derivative of a type, we don't differentiate the values to the left of the hole from the values to the right.

Take for example a $List$.

$$\partial List\ a = (List\ a, List\ a) \quad \text{-- the same as Zipper}$$

But for dissection, we note those values that have already been processed differently from those yet to be seen.

Dissection follows the same rules

Dissection follows the same rules.

$$
\begin{aligned}
\triangle K &= 0 \\
\triangle Id &= 1 \\
\triangle (p + q) &= \triangle p + \triangle q \\
\triangle (p \times q) &= \triangle p \times ⦣ q + \angle p \times \triangle q
\end{aligned}
$$

Dissection follows the same rules

Dissection follows the same rules.

$$
\begin{aligned}
\triangle\!\!\!\!\mid\;\, K &= 0 \\
\triangle\!\!\!\!\mid\;\, Id &= 1 \\
\triangle\!\!\!\!\mid\;\, (p + q) &= \triangle\!\!\!\!\mid\;\, p + \triangle\!\!\!\!\mid\;\, q \\
\triangle\!\!\!\!\mid\;\, (p \times q) &= \triangle\!\!\!\!\mid\;\, p \times \reflectbox{$\angle$}\, q + \angle\, p \times \triangle\!\!\!\!\mid\;\, q
\end{aligned}
$$

What is $\angle$ and $\reflectbox{$\angle$}$?

Dissection follows the same rules

Dissection follows the same rules.

$$
\begin{aligned}
\Delta K &= 0 \\
\Delta Id &= 1 \\
\Delta (p + q) &= \Delta p + \Delta q \\
\Delta (p \times q) &= \Delta p \times \reflectbox{$\angle$} q + \angle p \times \Delta q
\end{aligned}
$$

What is $\angle$ and $\reflectbox{$\angle$}$?

$\angle$ labels the values in the type as having been processed,

Dissection follows the same rules

Dissection follows the same rules.

$$
\begin{aligned}
\triangle\!\!\!\!\mid\, K &= 0 \\
\triangle\!\!\!\!\mid\, Id &= 1 \\
\triangle\!\!\!\!\mid\, (p + q) &= \triangle\!\!\!\!\mid\, p + \triangle\!\!\!\!\mid\, q \\
\triangle\!\!\!\!\mid\, (p \times q) &= \triangle\!\!\!\!\mid\, p \times \reflectbox{$\angle$}\, q + \angle\, p \times \triangle\!\!\!\!\mid\, q
\end{aligned}
$$

What is $\angle$ and $\reflectbox{$\angle$}$?

$\angle$ labels the values in the type as having been processed,

$\reflectbox{$\angle$}$ labels the values in the type as having *yet* to be processed.

Dissection of a List

What if we want to dissect a $List$? Remember the derivative.

$$\partial(List\ a) = (List\ a, List\ a)$$

Dissection of a List

What if we want to dissect a $List$? Remember the derivative.

$$\partial(List\ a) = (List\ a, List\ a)$$

It is the same as $Zipper$ but where we annotate the parts of the list we have already seen ($\angle(List\ a)$) and those we have yet to see ($\reflectbox{$\angle$}(List\ a)$).

$$\triangle(List\ a) = (\angle(List\ a), \reflectbox{$\angle$}(List\ a))$$

Dissection of a List

What if we want to dissect a $List$? Remember the derivative.

$$\partial(List\ a) = (List\ a, List\ a)$$

It is the same as $Zipper$ but where we annotate the parts of the list we have already seen ($\angle(List\ a)$) and those we have yet to see ($\text{⦣}(List\ a)$).

$$\triangle(List\ a) = (\angle(List\ a), \text{⦣}(List\ a))$$

If we had used the same label in the dissection, we get the derivative.

$$\triangle(List\ a) = (\angle(List\ a), \angle(List\ a)) \cong ((List\ a), List\ a)$$

Summary

What have we learned today?

- How to describe an algebraic structure $(\mathcal{S}, Op)$ and monoids.

Summary

What have we learned today?

- How to describe an algebraic structure $(\mathcal{S}, Op)$ and monoids.
- How $\cdot + \cdot$ and $\cdot \times \cdot$ each form monoids, and together form a closed semiring.

Summary

What have we learned today?

- How to describe an algebraic structure $(\mathcal{S}, Op)$ and monoids.
- How $\cdot + \cdot$ and $\cdot \times \cdot$ each form monoids, and together form a closed semiring.
- How to take the derivative of a type, giving a one hole context.

Summary

What have we learned today?

- How to describe an algebraic structure $(\mathcal{S}, Op)$ and monoids.
- How $\cdot + \cdot$ and $\cdot \times \cdot$ each form monoids, and together form a closed semiring.
- How to take the derivative of a type, giving a one hole context.
- How to take the dissection of a type, giving a one hole context parameterized with where you have been before.

Table of Contents

Identity Data Type

We can define an identity data type that only holds a single value.[7].

$$\textbf{data}\ Id_1\ x = Id_1\ x \quad \text{-- element}$$

We note that this can be a functor with the following definition.

$$\begin{array}{l}\textbf{instance}\ \mathit{Functor}\ Id_1\ \textbf{where} \\ \quad \mathit{fmap}\ f\ (Id_1\ x) = Id_1\ (f\ x)\end{array}$$

[7] All definitions are originally from [**?**]

Constant Data Type

$$\textbf{data}\ K_1\ a\ x = K_1\ a \quad \text{-- constant}$$

This is the same as the $const$ function. $const\ x :: b \rightarrow a$ is a function that ignores its input.

We note that this can also be a functor.

$$\begin{aligned}&\textbf{instance}\ \mathit{Functor}\ (K_1\ a)\ \textbf{where}\\&\quad \mathit{fmap}\ f\ (K_1\ a) = K_1\ a\end{aligned}$$

Pairs

Data types can be defined by the product of two other types. Often this appears as the following.

$$\textbf{data}\ Pair\ a\ b = Pair\ a\ b$$

We can write this generically as follows.

$$\textbf{data}\ p \times_1 q\ x = (p\ x\ , q\ x)_1 \quad \text{-- pairing}$$

And again we note that this can be a functor.

$$\begin{array}{l}\textbf{instance}\ (Functor\ p, Functor\ q) \Rightarrow Functor\ p \times_1 q\ \textbf{where} \\ \quad fmap\ f\ (p\ , q)_1 = (fmap\ f\ p\ , fmap\ f\ q)_1\end{array}$$

Sums

We can also construct a data type that chooses between one of two alternates.

$$\mathbf{data}\ \mathit{Either}\ a\ b = \mathit{Left}\ a \mid \mathit{Right}\ b$$

We can write this generically as follows.

$$\mathbf{data}\ p +_1 q\ x = L_1\ (p\ x) \mid R_1\ (q\ x) \quad \text{-- choice, aka Either}$$

And again we note that this can be a functor.

$$\begin{array}{l}
\mathbf{instance}\ (\mathit{Functor}\ p, \mathit{Functor}\ q) \Rightarrow \mathit{Functor}\ p +_1 q\ \mathbf{where} \\
\quad \mathit{fmap}\ f\ (L_1\ p) = L_1\ (\mathit{fmap}\ f\ p) \\
\quad \mathit{fmap}\ f\ (R_1\ q) = R_1\ (\mathit{fmap}\ f\ q)
\end{array}$$

One

If we think hard, we can think of a type that can only be constructed in one manner.[8]

$$\textbf{type } \mathit{Unit} = ()$$

This can be encoded in our generic patterns using K_1.

$$\textbf{type } 1_1 = K_1\ ()$$
$$1_v = K_1\ ()$$

[8] Remember that the only inhabitant of Unit is Unit. We ignore $\perp$.

Zero?

What if I want to define a type that cannot be inhabited at all? I would want something like the following.

$$\mathbf{data}\ 0 = ???$$

Well we can make an uninhabited type in the following way.

$$\mathbf{data}\ 0_1$$

This means we can never call the following function, which is probably a good thing. :-)

Recreating Maybe

Using this construction, we can recreate $Maybe$

$$\textbf{type}\ Maybe\ a = (1_1 +_1 Id_1)\ a$$

$$\begin{aligned} nothing &= L_1\ (K_1\ ()) \\ just\ x\quad &= R_1\ (Id_1\ x) \end{aligned}$$

Constructing a value is pretty simple.

$$val = just\ 5 :: Maybe\ Integer$$

And note that $Maybe$ already has a definition of $fmap$!

$$newVal = fmap\ (4+)\ val \quad \text{--} \equiv just\ 9$$

Back to Monoids

Remember that a monoid $(\mathcal{S}, Op)$ has to have a closed, associative binary operator with an identity element.

A *semiring* is when two monoids are defined over the same set, have two different identity elements, and one of the monoids commutes $(+)$. We write this as follows.

$$(\mathcal{S}, +, 1, \cdot, 0)$$

Product Types

Remember that the $\cdot \times_1 \cdot$ type is defined as follows.

$$\textbf{data}\ p \times_1 q\ x = (p\ x\ , q\ x)_1 \quad \text{-- pairing}$$

If we define the set of all Haskell types $\mathcal{H}$, then

List

If we attempt to make a list with this formulation, then we could write the following.

$$\textbf{type}\ ListF\ a = 1_1 +_1 K_1\ a \times_1 Id_1$$

We would like a list that contained $ListF$, but if we try

$$\textbf{type}\ List = ListF\ List$$

we get an infinite type. To fix this, we need to "tie the knot".

$$\textbf{data}\ \mu\, p = \hat{\mu}\, p\ (\mu\, p)$$

And now we can define our list as follows.

$$\begin{aligned}
\textbf{type}\ List\ \ a\ \ &= \mu\,(ListF\ a)\\
[\,]\ \ &= \hat{\mu}\,(L_1\ 1_v)\\
(:)\ a\ as\ \ &= \hat{\mu}\,(R_1\ (K_1\ a\,, Id_1\ as)_1)
\end{aligned}$$