The Algebra of Algebraic Data Types
Spare-Time Teaching

Mathias Vorreiter Pedersen

5 February 2016

Contents

1 Types as sets of values

1.1 Examples

$$\begin{aligned}
\mathsf{Bool} &= \{\mathsf{False}, \mathsf{True}\} \\
\mathsf{Int} &= \{-2^{31}, -2^{31}+1, \ldots, 0, 1, \ldots, 2^{31}-1\} \\
\mathsf{Bool} \to \mathsf{Bool} &= \{f, g, h, t\}
\end{aligned}$$

where

$$\begin{aligned}
f(\mathsf{False}) &= \mathsf{False} \\
f(\mathsf{True}) &= \mathsf{False}
\end{aligned}$$

$$\begin{aligned}
g(\mathsf{False}) &= \mathsf{False} \\
g(\mathsf{True}) &= \mathsf{True}
\end{aligned}$$

$$\begin{aligned}
h(\mathsf{False}) &= \mathsf{True} \\
h(\mathsf{True}) &= \mathsf{False}
\end{aligned}$$

$$\begin{aligned}
t(\mathsf{False}) &= \mathsf{True} \\
t(\mathsf{True}) &= \mathsf{True}
\end{aligned}$$

1.2 Constructing types

Given some base types T and U we can construct new types $T + U$ and $T * U$ as follows

$$\begin{aligned}
T * U &= \{(t, u) \mid t : T, u : U\} \\
T + U &= \{\mathsf{Left}\ t \mid t : T\} \cup \{\mathsf{Right}\ u \mid u : U\}
\end{aligned}$$

Product types ($T * U$) comes with a unit type 1 with a single value written as $()$. Similarly sum types ($T + U$) comes with a zero type 0 with no values. That is, $0 = \varnothing$.

Finally we can construct arrow types $T \to U$. Formally

$$T \to U = \{f \mid \mathsf{dom}(f) = T \wedge \mathsf{cod}(f) = U\}.$$

That is, the set of functions from set of values of type T to the set of values of type U.

1.2.1 Examples

$$
\begin{aligned}
1 + 1 &= \{\mathsf{Left}\ (), \mathsf{Right}\ ()\} \\
(\mathsf{Int}, \mathsf{Bool}) &= \{(-2^{31}, \mathsf{False}), (-2^{31}, \mathsf{True}), (-2^{31}+1, \mathsf{False}), (-2^{31}+1, \mathsf{True}), \\
&\qquad \ldots, (0, \mathsf{False}), (0, \mathsf{True}), \ldots, (2^{31}-1, \mathsf{False}), (2^{31}-1, \mathsf{True})\}
\end{aligned}
$$

1.3 Type isomorphisms

Question: Given two types T and U. When does the type T represent the same set as the type U?

Answer: When we can convert the values of one of the types to the other without losing any information! Formally: When there exists a bijection between the sets represented by the types.

For example: $1 + 1 \cong \mathsf{Bool}$ because there exists $f : 1 + 1 \to \mathsf{Bool}$ and $g : \mathsf{Bool} \to 1 + 1$ such that $\forall x : 1 + 1.\ g(f(x)) = x$ and $\forall x : \mathsf{Bool}.\ g(f(x)) = x$. Concretely:

$$
f(x) = \begin{cases} \mathsf{False} & \text{if } x = \mathsf{Left}\ () \\ \mathsf{True} & \text{if } x = \mathsf{Right}\ () \end{cases}
$$

$$
g(x) = \begin{cases} \mathsf{Left}\ () & \text{if } x = \mathsf{False} \\ \mathsf{Right}\ () & \text{if } x = \mathsf{True} \end{cases}
$$

We will explore this in great detail today.

2 Types as algebraic objects

The notation hints at some specific properties that we expect the sum and product types to uphold. Let's examine some of these.

2.1 Sum types

For natural numbers we know that

$$\forall n \in \mathbb{N}.\ n + 0 = n.$$

So we would also expect that for all types we have[1]

$$T + 0 \cong T$$

[1] We could quantify over all types as $\forall\ T : *$ since we only work with types of kind $*$ for now.

and indeed:

$$f : T + 0 \to T$$
$$f(\mathsf{Left}\ x) = x$$

and since 0 has no value we don't need to define f on values of type 0, since no such value exists. Similarly we define

$$g : T \to T + 0$$
$$g(x) = \mathsf{Left}\ x$$

and it's easily seen now that $\forall x \in T + 0$ we have $g(f(x)) = x$ and for all $x \in T$ we have $f(g(x)) = x$.

Similarly we may prove that $T + U \cong U + T$ by constructing functions f and g that swaps the left and right values, and likewise the associative property $(T + U) + W \cong T + (U + W)$ can be proven. Thus sum types really correspond to addition as one might know from ring theory.[2] Now for product types.

2.2 Product types

We would expect that $T * 1 \cong T$. We verify:

$$f : T * 1 \to T$$
$$f(x, ()) = x$$

and

$$g : T \to T * 1$$
$$g(x) = (x, ())$$

and the proof of $T * U \cong U * T$ is just the swap function! One can also easily prove $(T * U) * W \cong T * (U * W)$.

2.3 The distributive law

A more interesting case is the distributive law:

$$\forall a, b, c \in \mathbb{N}.\ a * (b + c) = a * b + a * c.$$

In the world of types this can be translated into the following statement:

For all types A, B, C we have the isomorphism

$$A * (B + C) \cong A * B + A * C$$

[2] Types actually form a semi-ring, since we have addition and multiplication with unit elements, but no additive nor multiplicative inverses.

and we prove this, as usual, by providing a functions

$$f : A * (B + C) \to A * B + A * C$$
$$g : A * B + A * C \to A * (B + C)$$

First we construct f.

$$f(a, \mathsf{Left}\ b) = \mathsf{Left}\ (a, b)$$
$$f(a, \mathsf{Right}\ c) = \mathsf{Right}\ (a, c)$$

and then g.

$$g(\mathsf{Left}\ (a, b)) = (a, \mathsf{Left}\ b)$$
$$g(\mathsf{Right}\ (a, c)) = (a, \mathsf{Right}\ c)$$

and verify (Using function composition and the polymorphic identity function for short notation) that $g \circ f = \mathsf{id}$ and $f \circ g = \mathsf{id}$.

2.4 Function types

So sum types are $+$, product types are $*$. What are function types? To answer this let's count the number of values of type $A + B$, and the number of values of type $A * B$. By definition

$$\begin{aligned} |A + B| &= |\{\mathsf{Left}\ a \mid a : A\} \cup \{\mathsf{Right}\ b \mid b : B\}| \\ &= |\{\mathsf{Left}\ a \mid a : A\}| + |\{\mathsf{Right}\ b \mid b : B\}| \\ &= |A| + |B| \end{aligned}$$

and similarly $|A * B| = |A| * |B|$.

Keeping this in mind. What is $|A \to B|$? That is, how many functions $f : A \to B$ exists?

For each $a \in A$ $f(a)$ can be one of $|B|$ possible values. and since there are $|A|$ values in the domain, this means that there are $|B|^{|A|}$ possible functions with domain A and codomain B.

So if function types correspond to exponentiation, one would expect the identity

$$\forall a, b, c \in \mathbb{N}.\ c^{a+b} = c^a \cdot c^b$$

or in the world of types:

$$A + B \to C \cong (A \to C) * (B \to C)$$

and indeed:

$$f : (A + B \to C) \to (A \to C) * (B \to C)$$
$$f(h) = (\lambda\, a.h(\mathsf{Left}\ a), \lambda\, b.h(\mathsf{Right}\ a))$$

and

$$g : (A \to C) * (B \to C) \to (A + B \to C)$$
$$g(a2c, b2c) = \lambda\, x.\ \begin{cases} a2c(a) & \text{if } x = \mathsf{Left}\ a \\ b2c(b) & \text{if } x = \mathsf{Right}\ b \end{cases}$$

Which one can easily prove to satisfy $f \circ g = \mathsf{id}$ and $g \circ f = \mathsf{id}$. Another interesting identity is

$$\forall a, b, c \in \mathbb{N}.\ (c^b)^a = c^{a*b}$$

which looks more familiar in the world of types.

$$A \to B \to C \cong A * B \to C.$$

Oh look! Currying!

3 Recursive data types

3.1 Lists

One of the simplest recursive data type possible are lists. Given a type a. The list type $L(a)$ is defined as

$$L(a) = 1 + a * L(a)$$

That is. The type $L(a)$ is the type satisfying the isomorphism

$$L(a) \cong 1 + a * L(a).$$

Or alternatively: It's a fixpoint of this equation. We can find this fixpoint by infinitely replacing the $L(a)$ on the right-hand side by the definition of $L(a)$. Doing this, we get

$$\begin{aligned} L(a) &\cong 1 + a * L(a) \\ &\cong 1 + a * (1 + a * L(a)) \\ &\cong 1 + a + a^2 * L(a) \\ &\cong 1 + a + a^2 * (1 + a * L(a)) \\ &\cong 1 + a + a^2 + a^3 * L(a) \\ &\dots \\ &\cong 1 + a + a^2 + a^3 + a^4 + \dots \end{aligned}$$

That is, the list type is the set consisting of the unit value, a value consisting of two values, a value consisting of three values, etc. This is exactly how we think

of lists! But that's not surprising since we just unrolled the definition infinitely many times. We can do something much *much* cooler!

Let's think of

$$L(a) \cong 1 + a * L(a).$$

as the equation

$$L(a) = 1 + a * L(a)$$

and let's solve for $L(a)$. Doing this we get:

$$\begin{aligned} L(a) = 1 + a * L(a) &\Leftrightarrow L(a) - a * L(a) = 1 \\ &\Leftrightarrow L(a) * (1 - a) = 1 \\ &\Leftrightarrow L(a) = \frac{1}{1-a} \end{aligned}$$

What? The list type is $1/(1-a)$? What does division of types mean? And what does negative types mean? I don't know![3] Anyway we don't really need to know what it means, because we have calculus!

Remember Taylor's theorem? A special case of his theorem ($a = 0$, also called the Maclaurin series) says that a function $f : \mathbb{C} \to \mathbb{C}$ can be described as an infinite sum of derivatives as follows

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(0)}{n!} x^n$$

performing this trick on our weird definition of $L(a) = 1/(1-a)$ we get

$$L(a) = \frac{1}{1-a} = 1 + a + a^2 + a^3 + a^4 + \ldots$$

Yes, really. We just derived that a list is a sequence of values of arbitrarily length - using mathematics from the 17th century! But wait! There's more!

3.2 Binary trees

A binary tree can be described using the following recursive data type.

$$T(a) = 1 + a * T(a)^2$$

Let's apply the same idea as before: We'll solve for $T(a)$ and use Taylor's expansion to get a result that we can hopefully interpret.

Being a quadratic equation, we can rewrite and solve as follows

$$\begin{aligned} T(a) = 1 + a * T(a)^2 &\Leftrightarrow 0 = 1 - T(a) + a * T(a)^2 \\ &\Leftarrow T(a) = \frac{1 - \sqrt{1 - 4a}}{2a} \end{aligned}$$

[3] Okay, I have an idea. But I won't tell you right now.

Note that we ignore the other solution to the equation, since a quadratic formula has more than one solution. But this other solution is more difficult to give a meaningful interpretation to. If we ask Wolfram Alpha to Taylor expand this formulla we get

$$T(a) = 1 + x + 2x^2 + 5x^3 + 14x^4 + \ldots$$

That is, a binary tree is either empty, a tree consisting of a single root, one of two possible trees with 2 nodes, one of five possible trees with 3 nodes, etc. Thus by looking at the coefficients of this infinite polynomial we've derived a formula for the number of binary trees with any given number of nodes. In other words: We've used types and calculus to derive one of the most important series in combinatorial mathematics: Catalan numbers!

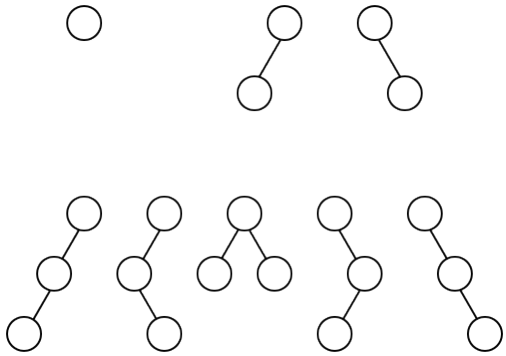

Figure 1: All binary trees with 0 nodes, 1 node, 2 nodes and 3 nodes.

4 Derivatives of algebraic data types

4.1 Zippers

We will now give an interpretation of the derivative of an algebraic data type, and show how all of the rules you know and love apply to our scenario. Before doing this, we need to do some coding! Specifically we introduce the concept of a Zipper.

This is a very general concept, but we'll do it only for the case of lists. Thus given the following definition of a polymorphic list type

```
data List a
  = Nil
  | Cons a (List a)
  deriving Show
```

we define the type `Zip a` consisting of a single a value and a context with a single hole.

```
data Zip a
  = Zip (List a) a (List a)
  deriving Show
```

A zipper comes with two functions for navigating the data type:

`left :: Zip a -> Zip a` and `right :: Zip a -> Zip a` for moving the focus point of the zipper one step left or one step right respectively. For polymorphic lists we can define these as follows.

```
left :: Zip a -> Zip a
left (Zip (Cons x xs) a ys) = Zip xs x (Cons a ys)

right :: Zip a -> Zip a
right (Zip xs a (Cons y ys)) = Zip (Cons a xs) y ys
```

and for convenience we have another function for extracting the focus point.

```
focus :: Zip a -> a
focus (Zip _ a _) = a
```

Below is an example program using `left` and `right`.

```
main = do
  print (focus z) -- prints 0
  print (focus . left $ z) -- prints 1
  print (focus . left . left $ z) -- prints 2
  print (focus . left . left . left $ z) -- prints 3
  print (focus . left . left . left . left $ z) -- prints 4
  where z = Zip (Cons 1 (Cons 2 (Cons 3 (Cons 4 Nil)))) 0 Nil
```

In the list example we call the type $L(a) * L(a)$ a context with a hole and say that a zipper is a type consisting of a context with a single hole and a focus point. The question is now: How do we go from an algebraic data type, to the type of the context, in a mechanical way?

4.2 Derivatives

Let's say we have some type variable a. For instance $a = \mathsf{Bool}$ or $a = \mathsf{Int}$. What's the one-hole context for such a type a? There's only one place to put the hole, so we don't need any context to place it. Thus the context here is the type 1.

What if we wanted the one-hole context of the type $a^2 = a * a$. There can only be one hole in the type, so either the hole is in the first component, or it's in the second component.

Thus the type of the context must be $a + a \cong 2 * a$. The type 2 tells us which component the hole is in, and the value of type a is then the remaining context for either the left or the right part of the sum type.

What about $a^3 = a * a * a$? The hole can be in one of three possible places, and in every case we need to remember the other two values, so the type will be $a^2 + a^2 + a^2$, or $3 * a^2$.

Generalizing this a bit, we see that the one-hole context of the tuple-type a^n is $n * a^{n-1}$. Looks a lot like the power rule, right? Let's use the familiar notation ∂a to denote the one-hole context of the type a and let's explore other familiar rules from differentiation.

What is the one hole context for the type 1? Well, that makes no sense since the only possible value is (), so there is no hole. Thus $\partial 1 = 0$.

What about the one-hole context of a sum type $a + b$? Either the hole is on the left side, or it's on the right side. So $\partial(a + b) = \partial a + \partial b$. Just like Newton said!

What about product types? Given a pair $a * b$. Either we can put the hole in the first component, and thus we need to keep the second component. Alternatively we can put the hole in the second component, and thus we need to keep the first component. Therefore

$$\partial(a * b) = \partial f * g + f * \partial g.$$

What about the chain rule? Given a polymorphic type $\forall a.T$, instantiated with a concrete type a, what is the one-hole context of $T(a)$? In this case we need to keep track of two holes: One for the outer type T, and one for the inner type a (Think of lists of lists for instance!). Thus

$$\partial(T(a)) = \partial T(a) * \partial a$$

Having given an interpretation to all of the standard rules for derivatives, let's revisit the example of lists. Recall that

$$L(a) = \frac{1}{1 - a}$$

and thus

$$\partial(L(a)) = \partial L(a) * \partial a = \frac{1}{1 - a} * 1 = \frac{1}{(1 - a)^2} = \left(\frac{1}{1 - a}\right)^2 = L(a)^2$$

4.3 Deriving a zipper for binary trees

Recall that the type of a binary tree is

$$T(a) = 1 + a * T(a)^2.$$

We apply implicit differentiation and solve for $\partial T(a)$ and get

$$\begin{aligned}
\partial(T(a)) = \partial 1 + \partial(a * T(a)^2) &\Leftrightarrow \partial(T(a)) = \partial(a * T(a)^2) \\
&\Leftrightarrow \partial(T(a)) = T(a)^2 + 2 * a * T(a) * \partial(T(a)) \\
&\Leftrightarrow 0 = T(a)^2 + 2 * a * T(a) * \partial(T(a)) - \partial(T(a)) \\
&\Leftrightarrow 0 = T(a)^2 + \partial(T(a)) * (2 * a * T(a) - 1) \\
&\Leftrightarrow \partial(T(a)) = \frac{T(a)^2}{1 - 2 * a * T(a)}
\end{aligned}$$

So the one-hole context for binary trees is $\frac{T(a)^2}{1-2*a*T(a)}$. But using the fact that $L(a) = \frac{1}{1-a}$ we can rewrite $\partial(T(a))$ as

$$\partial(T(a)) = T(a)^2 * L(2 * a * T(a))$$

This type may not make a lot of sense, but we can actually implement a zipper for binary trees using this data type.

In Figure 2 a binary tree is shown. The two blue subtrees represent the first $T(a)^2$ component of the pair. The circled trees represent the tree-component of the list in the zipper. Let $(b, x, t) : 2 * a * T(a)$ be an element of the list. The subtree t is one of the circled subtrees, and x is the root of the subtree t. The boolean b tells you whether the subtree is a left, or right child of another tree in the list.

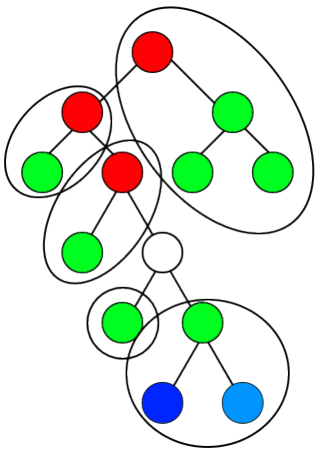

Figure 2: Interpretation of the one-hole context for binary trees

5 Non-algebraic data types: An example

Let's try an example on something that is not an algebraic data type: Sets. A set is not an algebraic data type because it imposes restrictions on the elements, apart from what we can express with the constructors like $+$ and $*$. For sets we have two properties: Uniqueness of elements, and irrelevance of order.

What is the type of a set that can contain zero elements, $\mathsf{Set}_0(a)$? That's clearly 1, since it carries no information. What about $\mathsf{Set}_1(a)$? Clearly the type of a set that can contain only one element must be equal to the type of the underlying type. So $\mathsf{Set}_1(a)$ has type a.

How many different sets exists of size two? That is, what is $\mathsf{Set}_2(a)$? For the first position there can be a different elements, and for the next position there can be $a-1$ different elements, since sets contain unique values. But now we over counted since the ordering of elements in a set doesn't matter (for instance the sets $\{a,b\}$ and $\{b,a\}$ represent the same set), so $\mathsf{Set}_2(a) = \frac{a\cdot(a-1)}{2}$. In general we have

$$\begin{aligned}\mathsf{Set}_n(a) &= \frac{a\cdot(a-1)\cdots(a-n+1)}{n!}\\ &= \frac{a^{\underline{n}}}{n!}\end{aligned}$$

So if we don't care about the size we can write the set type as an infinite sum type

$$\begin{aligned}\mathsf{Set}(a) &= \mathsf{Set}_0(a) + \mathsf{Set}_1(a) + \mathsf{Set}_2(a) + \mathsf{Set}_3(a) + \cdots\\ &= 1 + a + \frac{a^{\underline{2}}}{2!} + \frac{a^{\underline{3}}}{3!} + \cdots\end{aligned}$$

So understand this type a bit more we use a forward difference operator

$$\Delta\mathsf{Set}(a) = \mathsf{Set}(a+1) - \mathsf{Set}(a)$$

and we can now show that

$$\Delta\mathsf{Set}(a) = \mathsf{Set}(a)$$

and thus

$$\mathsf{Set}(a+1) - \mathsf{Set}(a) = \mathsf{Set}(a)$$

or equivalently

$$\mathsf{Set}(a+1) = 2\cdot\mathsf{Set}(a)$$

and thus $\mathsf{Set}(a)$ must satisfy

$$\mathsf{Set}(a) = 2^a$$

which in the world of types mean

$$\mathsf{Set}(a) \cong a \to \mathsf{Bool}$$

That is, a set is just a function mapping a value to a boolean value telling us whether the value is a member of the set!