Algebraic Datatypes and their Derivatives

Chris Grossack
150.5

Nov 4, 2017

1 Datatypes

Functional programming in SML allows for the creation of custom datatypes with the `datatype` keyword. In these custom datatypes, we can combine two types in the following ways:

```
datatype ('a, 'b) product = Pair of ('a * 'b)
datatype ('a, 'b) sum = Left of 'a | Right of 'b
```

The `product` datatype allows us to specify that we want to store both an α and a β. So for exapmle,

```
Pair (7,true) :  (int,bool) product
Pair ("hello", 42) :  (string,int) product
Pair (42, "hello") :  (int,string) product
```

It stores *both* an `int` and a `bool`.

Contrast this to the `sum` datatype, which allows us to store an α or a β. For example,

```
Left 7 :  (int,string) sum
Right "hello" :  (int,string) sum
Left true :  (bool, int) sum
Right 7 :  (bool, int) sum
Right false :  (int, bool) sum
Left 42 :  (int, bool) sum
```

Here we are storing *either* one of two types.

Note: from here forwards, I will write the type `('a,'b) sum` as $\alpha + \beta$, and `('a,'b) product` as $\alpha \times \beta$

2 Algebra

The fact that I'm using + and $\times$ to refer to these operations is a bit of a giveaway for this section. It turns out we can meaningfully do algebra with types, and this will be both useful and interesting. However, in order for many of the results to be true, we need to weaken our notion of equality. Rather than wanting two types to be exactly the same, we are interested in what information the type can contain. For instance, it is intuitive that storing `(5,(true, "hi"))` and `((5,true), "hi")` and `(true,(5,"hi"))` all have the same information. So we would like these types to be "equal". In pursuit of this, we make the following definition.

Definition 2.1. We say two types α and β are **isomorphic**, written $\alpha \cong \beta$, when there exist two functions $f : \alpha \to \beta$ and $g : \beta \to \alpha$ such that for all $a : \alpha$, $b : \beta$

- $g(f(a)) = a$
- $f(g(b)) = b$

These functions f and g allow us to convert between α and β, and the constraints on their compositions mean we cannot lose information when we convert in either direction. This means that the two types must carry exactly the same information, exactly like we wanted!

Now, for a brief formality:

Theorem 2.1. $\cong$ *forms an equivalence relation on types*

Proof. It suffices to show reflexivity, symmetry, and transitivity.

1: reflexivity ($\alpha \cong \alpha$)
(`fun id x = x`) : $\alpha \to \alpha$
works for both f and g, and has the desired properties.

2: symmetry ($\alpha \cong \beta$ implies $\beta \cong \alpha$)
Assume $\alpha \cong \beta$.
Then fix $f : \alpha \to \beta$ and $g : \beta \to \alpha$ by the definition of $\cong$.
Recognize that the same g and f still work for $\beta \cong \alpha$.

3: transitivity ($\alpha \cong \beta$ and $\beta \cong \gamma$ implies $\alpha \cong \gamma$)
Assume $\alpha \cong \beta$ and $\beta \cong \gamma$.
Then fix the following functions by the definition of $\cong$:

- $f_1 : \alpha \to \beta$
- $g_1 : \beta \to \alpha$
- $f_2 : \beta \to \gamma$
- $g_2 : \gamma \to \beta$

Now consider $f_2 \circ f_1 : \alpha \to \gamma$ and $g_1 \circ g_2 : \gamma \to \alpha$.
(where, as usual, $\circ$ denotes function composition)
fix $a : \alpha$ and $c : \gamma$

$$\begin{aligned}
(f_2 \circ f_1)((g_1 \circ g_2)(c)) &= f_2(f_1(g_1(g_2(c)))) && \text{defn function composition} \\
&= f_2(g_2(c)) && \text{since for all } b : \beta,\ f_1(g_1(b)) = b \\
&= c && \text{since for all } c : \gamma,\ f_2(g_2(c)) = c
\end{aligned}$$

$$\begin{aligned}
(g_1 \circ g_2)((f_2 \circ f_1)(a)) &= g_1(g_2(f_2(f_1(a)))) && \text{defn function composition} \\
&= g_1(f_1(a)) && \text{since for all } b : \beta,\ g_2(f_2(b)) = b \\
&= a && \text{since for all } a : \alpha,\ g_1(f_1(a)) = a
\end{aligned}$$

So $f_2 \circ f_1$ and $g_1 \circ g_2$ work, and $\cong$ is transitive.

Thus the theorem holds. □

Now that we have a notion of equality, let's prove some fundamental algebraic properties of these operations.

Proposition 2.1. $\alpha \times \beta \cong \beta \times \alpha$

Proof. The following two functions work.
`fun f (a,b) = (b,a)` : $\alpha \times \beta \to \beta \times \alpha$
`fun g (b,a) = (a,b)` : $\beta \times \alpha \to \alpha \times \beta$ □

Proposition 2.2. $\alpha + \beta \cong \beta + \alpha$

Proof. The following two functions work.

```
fun f (Left a) = Right a
| f (Right b) => Left b
fun g (Left b) = Right b
| g (Right a) => Left a
```

($f : \alpha + \beta \to \beta + \alpha$ and $g : \beta + \alpha \to \alpha + \beta$) □

Proposition 2.3. $\alpha \times (\beta + \gamma) \cong (\alpha \times \beta) + (\alpha \times \gamma)$

Proof. Again, the following functions work.

```
fun f (a,Left b) = Left (a,b)
| f (a,Right c) = Right (a,c)
fun g (Left (a,b)) = (a, Left b)
| g (Right (a,c)) = (a, Right c)
```

($f : \alpha \times (\beta + \gamma) \to (\alpha \times \beta) + (\alpha \times \gamma)$, and $g : (\alpha \times \beta) + (\alpha \times \gamma) \to \alpha \times (\beta + \gamma)$) □

Be sure you understand why the types of the above functions are correct, and verify that no information is lost during the conversions (by checking the conditions on the composition of f and g).

Exercise: Show that
$\alpha \times (\beta \times \gamma) \cong (\alpha \times \beta) \times \gamma$
$\alpha + (\beta + \gamma) \cong (\alpha + \beta) + \gamma$

Now that we know that multiplication and addition behave intuitively, we would like types **0** and **1** which serve as identities for + and ×.

1 happens to be `unit`, a type with exactly one value, typically denoted in sml by `()`. However, any type with exactly one value works, for instance `datatype meaningless = NIL` and `datatype halfabool = TRUE`. The fact that **1** ≅ `meaningless` ≅ `halfabool` is clear.

Proposition 2.4. ***1*** *works as a multiplicative identity. Namely,* $\alpha \times \mathbf{1} \cong \alpha$ *for all* α.

Proof. The following work.

```
fun f (a,()) = a
fun g a = (a,())
```

□

We also have $\mathbf{0}$, which carries no information at all. sml has no built in empty type, but we can create with the following trick:

```
datatype ZERO = Zero of ZERO
```

In order to create a value of type `ZERO`, we need to put a preexisting value of type `ZERO` inside it. However, since there is no base case, (and sml only allows finitely many type constructors) we have no way of doing so. Thus the type is empty, namely there is no value of type `ZERO`.

Exercise: show that[1]
$\alpha + \mathbf{0} \cong \alpha$
$\alpha \times \mathbf{0} \cong \mathbf{0}$.

Now, we know that any type with exactly 1 value is isomorphic to $\mathbf{1}$, but it would make a lot of sense to define $\mathbf{2}$ as $\mathbf{1} + \mathbf{1}$. This turns out to be isomrophic to any type with exactly two values. Recalling the implemenation of $\alpha + \beta$ as `('a, 'b) sum`, we can see those two values are `Left ()` and `Right ()`. However, we can pick any two values we like. For instance, `bool` $\cong \mathbf{2}$.

In fact, for any type with finitely many (say n) values, we can write is as
$\mathbf{n} \cong \underbrace{\mathbf{1} + \mathbf{1} \ldots + \mathbf{1}}_{n \text{ times}}$

Warning: Be careful not to confuse the *value* `n : int` and the *type* $\mathbf{n}$.

Finally for a brief example, recall the `'a option` type, which represents possible failure of a computation by adding an extra value called `NONE`.

```
datatype 'a option = NONE | SOME of 'a
```

Notice that an `'a option` is one of two things. The left thing, `NONE` has only one value it can take, whereas the right thing, `SOME of 'a` can take any value in α. So we can write α `option` $\cong \mathbf{1} + \alpha$.
Exercise: prove it

This representation looks awfully like the polynomial $1 + x$, which leads us into the following section.

[1] note: this will be somewhat difficult, as SML has no good way of working with empty types. As such, argue to yourself why it must be true using the idea of "$\mathbf{0}$ has no values inside it". Vigorous hand waving is acceptable here.

3 Polynomials

We know how to represent types with finitely many values now, but what about real datastructures, which hold values of other types? Let's consider the list, which you may recall is implemented as follows.

```
datatype 'a list = nil | Cons of ('a * 'a list)
```

Now, intuitively, what is a list? A list is either `nil`, which stores no information, *or* it is one element of type α *or* it is two elements of type α *or* it is three elemetns ....

This goes to say that one way of writing a list which holds αs as the following infinte sum: $L(\alpha) = 1 + \alpha + \alpha \times \alpha + \alpha \times \alpha \times \alpha + \dots$
If we abbreviate $\alpha \times \alpha$ in the obvious way, then we retrieve $L(\alpha) = 1 + \alpha + \alpha^2 + \alpha^3 + \dots$ which looks very much like a polynomial in α.

But how do we actually define lists? Well, a list is either empty, **1**, or it's a product of an α an another list. Namely: $L(\alpha) = \mathbf{1} + \alpha L(\alpha)$. But then, we can solve for $L(\alpha)$!

$$L(\alpha) = \frac{\mathbf{1}}{\mathbf{1} - \alpha}$$

This is utterly meaningless, since we don't know what $-\alpha$ means, and we CERTAINLY don't know what a fractional type means. That said, upon remembering calculus we start to see some glimmer of meaning, since (you may recall) the taylor expansion of $\frac{1}{1-x}$ is $1 + x + x^2 + x^3 + \dots$, the exact expression we intuited a list to be!

In fact, we can apply this methodology to all sorts of datatypes! Recall trees, defined as follows:

```
datatype 'a tree = Empty | Node of ('a tree * 'a * 'a tree)
```

Then a tree is either empty (**1**) or the product of two trees and an α, so $T(\alpha) = \mathbf{1} + \alpha T^2$. Solving in the same illegal fashion yields

$$T(\alpha) = \frac{\mathbf{1} \pm \sqrt{\mathbf{1} - \mathbf{4}\alpha}}{\mathbf{2}\alpha}$$

Depending on your background in combinatorics, you may or may not recognize this as the generating function for the **Catalan numbers**, which counts the number of ways to make a binary tree ...

In fact, taking the taylor expansion of this polynomial gives

$$T(\alpha) = \mathbf{1} + \alpha + \mathbf{2}\alpha^2 + \mathbf{5}\alpha^3 + \dots$$

And one can verify that these coefficients are exactly the number of possible binary trees. The α^0 term is **1**, since there is exactly one way to make an empty tree. Similarly there is only one way to make a tree storing only one α, namely `Node(Empty, x, Empty)`. Similarly, there are two ways to make a tree hold two

values of type α. One value goes at the root, and the other goes in either the left or right child. Continuing the pattern, there are five ways to create a tree storing three values, and in general, the coefficient of the α^n term tells you how many ways there are to store n many values in your tree.

Thinking about what this means as types, we can store every tree as *either* $\mathbf{1}$, representing empty, or α representing the unique tree with one element, or $\mathbf{2} \times \alpha^2$, representing a tuple of two αs and one "indicator". The indicator is an element of $\mathbf{2}$, and therefore has one of two values, representing each of the two possible tree shapes we could be storing our αs in. Similarly, for any term $\mathbf{C}_n\alpha^n$, we store the n many αs which represent the data in the tree, and we store an indicator coming from $\mathbf{C}_n$ telling us which of the C_n many tree shapes we are in. (here C_n is the nth catalan number).[2]

This brings us to the first major point of the talk:

> polymorphic datatypes are just polynomials over our regular types!

Where by "regular" I mean the types which look like natural numbers $\mathbf{n}$.[3] Finally, when we have polynomials, and we've been doing a bunch of seemingly illegal operations and getting meaningful results, we might try pushing our luck and taking a derivative of some datatype's polynomial and seeing what we get out ... But first, a diversion.[4]

[2]Clearly there is something meaningful here, despite the fact that manipulating these polynomials in this way seems to be getting less and less legal. What does it mean to take the square root of a type, as we did above? The programming implications of extending types to allow negative and fractional types is a current area of research, on the very edge of our type theoretic knowledge! I encourage you to read about it if you're interested!

[3]It's a fairly natural question to ask how polymorphic datatypes holding multiple types can be represented. Intuitively, they are just polynomials with multiple variables, however formalizing this concept involves quite a bit of algebra (really, we are defining a polynomial in β whose coefficients, rather than being numbers, are polynomials in α). The topic is incredibly interesting, but beyond the scope of this talk.

[4]These "illegal" operations are actually somewhat justified. There is a paper titled *Objects of Categories as Complex Numbers* which explains exactly why we are able to legally use these operations. It's a surprisingly easy paper to understand, and I encourage you to read it if you're interested!

4 One Hole Contexts

A lot of functional datastructures have constant time access near the outer layer of that structure, for instance the head of a list or the root of a tree. However, access at some random point inside the structure is typically linear in the number of constructors required to get there. For instance, looking at some element of a list is linear in the length of the list, and looking at some element of a tree is linear in the depth of the tree.

From a purely practical perspective, it would be useful to have a datastructure which stores similar information to that of a simpler datastructure, but which allows constant time access to some internal "hole" where we can put new data. It would also be useful to be able to move this "hole" around the datastructure quickly.

So, what exactly am I talking about? Consider the following representation of a humble list:

```
[1,2,3,[ ],4,5]
```

We have a hole in the center, where we can add new information in constant time. We might represent this datastructure as the following:

```
type 'a zipper = 'a list * 'a list
```

Here, we define an `'a zipper`, which is the traditional term for a one-hole context for lists. We use the left list to hold values to the left of the hole, and we use the right list to store values to the right of the hole. For example, the above list would be represented as follows:

```
([3,2,1],[4,5])
```

Notice we reverse the order of the left list so that we have constant time access to the region next to the hole.

To briefly explain how one might use the datatype, consider the following functions:

```
fun insert x (left,right) = (left,x::right)
fun moveLeft (l::left,right) = (left, l::right)
fun moveRight (left,r::right) = (r::right, left)
```

Notice the movement functions are partial. If you try to move left and there's nowhere to go, we raise an error.

Exercise: fix this by making moveLeft/moveRight return α `zipper option`.

Now, how might one create a similar one-hole context, but for trees? We want some way to fill a hole inside a tree, and move that hole up, or to either child. There is no obvious way to do so. Thankfully . . .

5 Datatype Derivatives

The fact that I included the previous section has probably given away the punchline, but let's move towards it anyways.

Recall that polymorphic datatypes are polynomials, and recursive datatypes are those where we can write an equivalence of polynomials. For instance

$$L(\alpha) = \mathbf{1} + \alpha L(\alpha)$$

which we can pretend is a regular polynomial, so we can factor and divide, leading to:

$$L(\alpha) = \frac{\mathbf{1}}{\mathbf{1} - \alpha}$$

Well, let's push our luck. Depending on how well you remember calculus, the following may require some revision (it did for me ...)

$$\frac{dL}{d\alpha} = \frac{\mathbf{1}}{(\mathbf{1} - \alpha)^2} = \left(\frac{\mathbf{1}}{\mathbf{1} - \alpha}\right)^2 = L^2$$

So we took the derivative of lists, and wound up with precisely its associated one-hole context! This is an extremely non-obvious result, which happens to be true in general.[5]

So then to find the datatype for a one-hole context for trees, it suffices to take the derivative of $T(\alpha) = \mathbf{1} + \alpha T^2$! Don't forget the product rule ...

$$\frac{dT}{d\alpha} = \alpha \mathbf{2} T \frac{dT}{d\alpha} + T^2$$

So, rearranging

$$\frac{dT}{d\alpha} = \frac{T^2}{\mathbf{1} - \mathbf{2}T\alpha} = T^2 \left(\frac{\mathbf{1}}{\mathbf{1} - \mathbf{2}T\alpha}\right) = T^2 L(\mathbf{2}T\alpha)$$

So we can see that our derivative of trees is the product of two trees and a list which contains a product of $\mathbf{2}$, a tree, and an α. Let's unpack that.

Recall that $\mathbf{2}$ is isomorphic to any type with two values. We will choose `datatype TWO = LEFT | RIGHT` for reasons which will soon become apparent.

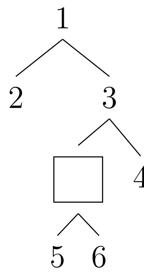
---

[5]For an outline of the proof of this claim, recognize every polynomial looks like a (possibly countably infinite) sum of terms which look like $\mathbf{a}_n \alpha^n$.

So we can first prove that the derivative of α^n is its one-hole context, then we can prove that constants $\mathbf{a}$ have derivatives corresponding to their one-hole context, then it remains to show that adding and multiplying these terms preserves the fact that derivatives correspond to a one-hole context.

This is a surprisingly simple proof to do, and I encourage you to give it a go!

Let's consider the following tree, with a hole in the marked position.



We would represent this tree with the following code:

```
datatype 'a tree = Empty | Node of ('a tree * 'a * 'alpha tree)
datatype TWO = LEFT | RIGHT
type 'a tree-with-hole =
'a tree * 'a tree * (TWO * 'alpha * 'a tree) list


val that_tree :  'a tree-with-hole =
(Node(Empty,5,Empty), Node(Empty,6,Empty),
[(RIGHT, 3, Node(Empty,4,Empty)), (LEFT, 1, Node(Empty,2,Empty))])
```

So we store a 3 tuple, the first two elements are the children of the hole, in this case the trees storing 5 and 6. The third element is a list representing the parents. So the first element represents the immediate parent, It stores a direction (which was our representation of **2**), a tree, and a value of type α. These three things represent half of a tree. The α represents the value stores in the node, the tree represents the other child, and the direction represents which side the other child belongs on. That is why our first parent stores `RIGHT`, `3`, and `Node(Empty,4,Empty)`. `Node(Empty,4,Empty)` is the `RIGHT` child of `3`. What is the left child? Well the left child is the hole, which we're storing outside the list!

Exercise:
Write a function `toContext :  'a tree -> 'a tree-with-hole` which creates a context where the cursor is at the root.

6 Application: Filesystem

Let's try to solve the same problem, but on rose trees instead of binary trees. A *rose tree* is a tree who has n many children, for any n. Consider the following implementation:

```
datatype 'a rtree = RNode of 'a * 'a rtree list
```

Notice we exclude the empty tree. We can show a node has no children by making its list of children empty. Also, for the purposes of this example, we will never deal with a truly empty tree. So we can write

$$R(\alpha) = \alpha L(R(\alpha))$$

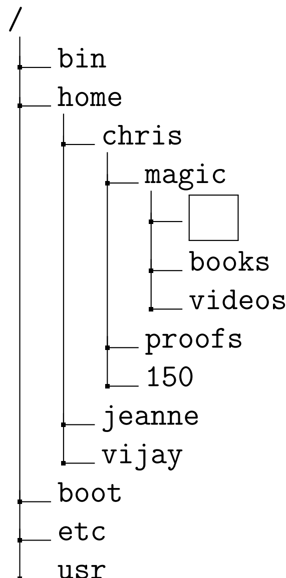
Differentiating, we obtain (recalling the product and chain rules. . .)

$$R'(\alpha) = L(R(\alpha)) + \alpha L'(R(\alpha))R'(\alpha)$$

Exercise: (challenge) Make sense of this

Intuitively, this should give us a structure which puts a hole somewhere inside of a rose tree. Then we should be able to move this hole up to a parent node, or down to any of the children. . .

If I simply add a suggestive diagram, then the structure we've created becomes clear.



Where now, we have constant access to one particular region of a tree, and constant time motion to move the hole to one of its children, or to its parent. Operations which are starting to sound an awful lot like `cd` . . . We appear to have accidentally made a filesystem. Who says theory is useless!