Algebra, Calculus, Types and Trees

Consider the type of finite binary trees (with no data at the nodes):

```
type t = Empty
       | Node of t * t
```

As usual, a binary tree is recursively defined to be either empty, or a node with a *left* subtree and a *right* subtree, which must both be (possibly empty) binary trees under this definition.

Next, consider the following function:

```
(* f: (t * t * t * t * t * t * t) -> t *)
let f (t1, t2, t3, t4, t5, t6, t7) =
  match (t1, t2, t3, t4) with
  | (Empty, Empty, Empty, Empty) ->
    begin
      match t5,t6 with
      | Node (t5a, t5b),_ -> Node(Node(Node(Node(Empty,t7),t6),t5a),t5b)
      | Empty, Node _ -> Node(Node(Node(Node(Node(t6,t7),Empty),Empty),Empty),Empty)
      | Empty, Empty _ ->
          match t7 with
          | Node(Node(Node(Node(t7a,t7b),t7c),t7d),t7e) ->
              Node(Node(Node(Node(Node(Empty,t7a),t7b),t7c),t7d),t7e)
          | _ -> t7
    end
  | _ -> Node(Node(Node(Node(Node(Node(Node(t7,t6),t5)),t4),t3),t2),t1)
```

It turns out that this function is actually extremely remarkable. Take a second and try to determine why.

In fact, this is an *injective* function from the type of 7-tuples of binary trees to single binary trees (it's actually bijective, but I won't prove that here). This is not, in and of itself, particularly alarming – there are countably infinite binary trees, and countably many n-tuples thereof (for finite n), so we could always obtain such an objection by way of encodings to and from the natural numbers. What is remarkable about *this* function, however, is that it only examines its arguments to a bounded, finite depth (namely, no tree is examined more than four levels deep).

Another interesting fact is that 7 is the *smallest* nontrivial tuple size for which this is possible (try it!). Nor will you be able to do this for tuples of size 8, 9, 10... but you can for 13, by using the above function twice. And similarly for any number that is 1 mod 6.

So how was this function produced?

Algebraic Datatypes

Products

Recall that we notate "tuple" types in ML as "multiplied", as in `a * b`. This is no accident, as in the lingo we refer to "tuple types" as "product types", as in *Cartesian product*, defined by the following rules:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash \pi_1\ e : \tau_1} \quad \frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash \pi_2\ e : \tau_2}$$

The first is an *introduction form*, which allows us to *create* a tuple. The others, then, are *elimination forms*, giving us mechanisms to get information *out* of a tuple. When programming in ML or Haskell, it is often preferred to use pattern matching instead of these *projection* operators, but in the underlying theory we consider these to be primitive (leaving pattern matching for another construct to be shown).

If you are good at nitpicking definitions, you may spot a problem with this setup – how do we construct tuples of size greater than two?

There are three reasonable ways to interpret the type $\tau_1 \times \tau_2 \times \tau_3$:

- Left-associative – $\tau_1 \times \tau_2 \times \tau_3 \triangleq (\tau_1 \times \tau_2) \times \tau_3$. No new constructs are needed; we can use π_1 and π_2 normally. On the other hand, indexing into the beginning of an n-tuple gets very irritating very quickly.
- Right-associative – $\tau_1 \times \tau_2 \times \tau_3 \triangleq \tau_1 \times (\tau_2 \times \tau_3)$. Similar to the left-associative version, but indexing into the *end* of a tuple is instead the irritating part.
- Non-associative – $\tau_1 \times \tau_2 \times \tau_3$ is primitive, and is effectively a macro over some other, n-ary type operator. This is the most pleasant to actually program with, but has a number of unpleasant consequences for the underlying typesystem. The π_1 operator can now work on many different types (really, any tuple type!), making the rules for the elimination form more complex. Similarly, these operators can no longer be safely used as first-order functions without introducing some form of ad-hoc polymorphism (what is the input type of *map* π_1?).

For the purpose of this class, we will take the third option and gloss over the details as usual.

You may also have heard of "record types", tuples with labeled elements. These are equivalent to standard tuples, but their elimination forms are instead notated π_ℓ, where ℓ is the label for the desired element (in fact, languages like Haskell treat records as syntactic sugar over records, in which π_ℓ is a macro for the corresponding π_i projection). Including these effectively locks us into the third interpretation of tuples, for the same reasons.

Closely related to the concept of product types is the so-called `unit` type, the type of the empty tuple. This type is often notated $\mathbf{1}$ (or just 1), as it has exactly one element, `()`. Just as the integer 1 is the identity for numerical multiplication, we often treat `unit` as the identity for multiplication of types as well, as we can transform `x : a` into `(x,()) : a * unit` and vice versa.

Sums

A difficulty that plagues many popular programming languages today is handling functions that can return multiple different types dependent on some condition. For example, a string processing function may want to return some deserialized data structure or an error message (a string). C handles this with "return codes", mapping each possible error message to an integer, but this doesn't generalize. In a dynamically typed language, you can avoid this by not writing down a return type at all, but this brings its own problems – in python, for example, library authors often have to write redundant type checks into their functions to handle a dynamic input type[1].

One idea might be, when returning something of type τ_1 *or* τ_2, to use some kind of product $int \times \tau_1 \times \tau_2$. In this encoding, the first element of the 3-tuple should be 0 or 1, and tells you which of the other projections are the "real" value, and that the other value is merely a sentinal. For example, to return an int that could also be a string, we might return the value `(0, 3, "")` – `0` to mean that we're returning an int, `3` is the "actual" return value, and `""` is just to fill out the tuple. This is (an abstract description of) the solution used in Java. However, this doesn't scale well. For one, we have to *store* a value of every possible return type, most of which are not used. A smarter idea is to use something like a C union along with a tag, where we only store the variant that we actually have.

This leads us to the variant/enum types found in ML and ML-based languages. In a type theory, these are *sum* types, dual to the aforementioned products. Sums are defined with the following introduction/elimination forms:

[1] This is alleviated to some extent by duck typing and an implicit contract with the caller, but this gets very messy very quickly if the shared behavior is not primitive. Python's `pathlib` library, for example, needs to have the path concatenation operator work on its own internal `Path` datatype (containing some extra metadata) as well with regular `str`s, with different merging behaviors for each.

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \texttt{INL}\{\tau_1 + \tau_2\}\ e : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \texttt{INR}\{\tau_1 + \tau_2\}\ e : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \Gamma, x_2 : \tau_1 \vdash e_2 : \tau}{\texttt{case}\{\tau_1 + \tau_2\}(e, x_1.e_1, x_2.e_2) : \tau}$$

A few notes. First, in the theory, we generally stick to binary sums, with variants `INL` and `INR`. When extending to a general-purpose programming language, on the other hand, we can amend the rules to allow for arbitrarily-named constructors, at the cost of some extra complexity. Similarly, the `case` construct in the elimination form is shorthand for the full `case` (or `match`, etc) expression, which can also be extended to account for arbitrary constructors (instead of left/right).

Just as `unit` is the identity for products on types, there is also a type notated $\mathbf{0}$, the identity on sums. This is `void` (not to be confused with the `void` found in languages like C or Java, which is actually `unit`!), the type with no members. To see this, notice that, as there is no way to construct a value of type `void`, any member of type $0 + \tau$ must have the form `INR e`, where `e` has type τ.

Polynomial Datatypes

With these, we now have enough background to show how to fit seven trees into one.

Types formed only from sums, products and type variables are known as *algebraic* datatypes, or a *polynomial* type/type operator.

Let's look at the definition of binary trees again.

```
type t = Empty
       | Node of t * t
```

With some sleight of hand, we can transmute it into a form that looks like what we had above:

```
type t = INL of unit
       | INR of t * t
```

Since `Empty` is a nullary constructor, we can attach a unit to it "for free". Then, we just rename the branches, giving us a polynomial type, which we can write using the shorter notation introduced earlier:

$$T = 1 + T^2$$

Read as a type isomorphism rather than a definitional equation, we say that we can transform back and forth between the types T and $1 + T^2$ "for free" – only

by wrapping constructors and pattern matching, and of course we can do this. Viewed as an *equation*, then, we get our first hint as to what's special about the number 7.

Solving this quadratic gives $T = \frac{1}{2} \pm i\frac{\sqrt{3}}{2}$. This is a primitive sixth root of unity, so we find that $T^6 = 1$, which is obviously false. Carrying forward, however, we conclude that $T^7 = T$, which is *not* obviously false, so it must be true.

Of course, this line of reasoning is completely bogus. For one, we went from a false statement ($T^6 = 1$ – "there is only one six-tuple of binary trees") to one we claim is true, namely that seven tuples of trees and singleton trees are isomorphic. Also, the "value" of T is completely nonsense – it contains fractional, negative and even *imaginary* types, none of which have any meaning in our type theory. And yet, the function from the beginning exists.

Type Algebra

Let's step back a bit. Remember that we defined two types to be "equal" algebraically when there is a "free" bijection between them. Then we can take the step

$$T^7 = T^8 + T^6$$

in a meaningful way: Consider one of the seven trees (the first, say) and set the others aside. This tree is either empty (leaving us only with the 6 remaining trees) or has two children (giving us the 6 remaining trees + 2 subtrees). And the reverse holds as well, where we can turn a structure that is either 6 trees or 8 trees into 7 trees. This can be iterated, ultimately leading to a step containing a 1, which cannot be reduced further. The full derivation can be found in the appendix.

Remember that we defined these equalities to be "free isomorphisms", namely those that can be written entirely via pattern matching or constructor application. The mystery of where the function from the beginning came from, then, is solved – by chaining free bijections together (and flipped left/right once, to make the code easier), we can determine what unique cases are necessary, and what they map to. Furthermore, because datatype constructors are finite, we know that the resulting function must also be "free".

Calculus

Arbitrary tree structures are mathematically important, but not particularly interesting from a programming perspective. We almost never want to be manipulating a tree or a list with no data in it.

Happily, though, we can easily extend our type system to add polynomial *type operators*, functions from types to types. Then, we can define the type of lists as

follows:

$$L(\alpha) = 1 + \alpha * L(\alpha)$$

derived from the constructors of list, `Nil` and `Cons`. So $L($`int`$)$ would be the type `int list`.

Now that we have things that look like functions on types (but not functions on values!), we may as well do something interesting with it. What if we take the derivative (I promise it's well-defined!)?

In order to do this, we're going to do something that isn't strictly allowed and solve for the closed form of the function L to find

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

In general, this wouldn't be well-formed due to the fractional and negative types. However, if we examine the Taylor series of this function (also found by repeatedly expanding the function L), we see

$$L(\alpha) = 1 + \alpha + \alpha^2 + \alpha^3 + \ldots$$

corresponding to the fact that a list of αs is 0- or 1- or 2- or so on many αs. This will generally work for recursive polynomial type operators, so we will work with the "invalid" representation for ease of manipulation.

Taking the derivative, we find that

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

Strangely, it seems that the derivative of a list is actually two lists. A coincidence?

One-hole contexts

One common problem faced by functional languages is that random access into a list is $O(n)$ in the index requested. In many workloads, however, it is unnecessary to have random access to many disparate elements of the list at once. It is often the case that most accesses are *spacially local*, or "close to each other". One idea, then, is to maintain a "cursor" into the list that allows constant time updating at the cursor, and constant time movement "left" or "right" on the list (think

about moving the cursor around in a text editor). You may have heard of this as a "zipper"[2].

What does this mythical data structure look like? Spiritually, for a list, a cursor needs two pieces of information only: The elements before the cursor, and the elements after. Thankfully, as the list interface gives constant time access to the first element and tail, this means that we can represent this zipper as two lists! But wait... isn't that what we said $L'(\alpha)$ is?

In fact, this works for any polynomial datatype you care to try. This will always give this kind of "cursor" structure, otherwise known as a *one hole context*, consisting of the information representing one structure with the type variable "removed" as the cursor.

Let's try a different structure. What about trees?

$$T(\alpha) = 1 + \alpha \times T^2(\alpha)$$

Deriving and solving, we get

$$T'(\alpha) = T^2(\alpha) + 2T(\alpha)T'(\alpha)$$

$$T'(\alpha) = T^2(\alpha) \times \frac{1}{1 - 2T(\alpha)} = T^2(\alpha) \times L(2T(\alpha))$$

Note that the type 2 here is $1 + 1...$ otherwise known as `bool`.

So what *would* a zipper on binary trees look like? To reconstruct a full tree from a given "cursor", you'd need:

- The children of the current node
- A list of ancestors of the current node

We'd actually need a bit more – we'd need to know whether each node is the left or right child of its parent.

Let's compare this with our derivative:

- Children of the current node: $T^2(\alpha)$, check.
- List of ancestors: $L(2 \times T(\alpha))$

And in fact, the list of ancestors *also* tracks whether the successor is a left or right child – that's where the 2 parameter comes in!

[2]You may protest that this is a job for regular random access arrays. You're right in the case that the workload does not require a lot of splitting or frequent addition/removals, all of which functional lists are extremely good at. Random access arrays are only good if the size of the array moves in one direction (generally growing, generally shrinking, or mostly constant).

References

- Mcbride, Conor. The Derivative of a Regular Type Is Its Type of One-Hole Contexts.
- Blass, Andreas. "Seven Trees in One." Journal of Pure and Applied Algebra, vol. 103, no. 1, Aug. 1995, pp. 1–21, arxiv.org/pdf/math/9405205.pdf, 10.1016/0022-4049(95)00098-h.

Appendix: Seven Trees

Here is the full derivation from T^7 to T, using only the semiring properties (associativity and commutativity of $+$ and $\times$) and the identity $T = T^2 + 1$.

$$
\begin{aligned}
T^7 &= T^8 + T^6 \\
&= T^8 + T^7 + T^5 \\
&= T^8 + T^7 + T^6 + T^4 \\
&= T^8 + T^7 + T^6 + T^5 + T^3 \\
&= T^8 + T^7 + T^6 + T^5 + T^4 + T^2 \\
&= T^8 + T^7 + T^6 + T^5 + T^4 + T^3 + T \\
&= T^8 + T^7 + T^6 + T^4 + T^4 + T \\
&= T^8 + T^7 + T^5 + T^4 + T \\
&= T^8 + T^6 + T^4 + T \\
&= T^7 + T^4 + T \\
&= T^7 + T^4 + T^2 + 1 \\
&= T^7 + T^4 + T^3 + T + 1 \\
&= T^7 + T^4 + T^4 + T^2 + T + 1 \\
&= T^7 + T^5 + T^4 + T^3 + T^2 + T + 1 \\
&= T^6 + T^4 + T^3 + T^2 + T + 1 \\
&= T^5 + T^3 + T^2 + T + 1 \\
&= T^4 + T^2 + T + 1 \\
&= T^3 + T + 1 \\
&= T^2 + 1 \\
&= T
\end{aligned}
$$

Notice that after reaching $T^7 + T^4 + T$„ we expand the T to get $T^2 + T^0$, preventing us from using the same trick to derive T from T^6.