Algebraic Data Types

1 Data Types

A data type is a class of values. For example 5, 3, etc are all of type `Int`. Values that belong to `Int` can be added to each other. Another data type is `String`, with string addition being defined as concataction. Note that there is no natural way to add an `Int` to a `String`.

A compound data type is a type that is built from a smaller type, or parameterized by another type. `[Int]` is a list of Ints, which is a list CDT parameterized with Int. We can, for example, `map +1 list_of_ints` which maps the `+1` function over a `[Int]`, and our typechecker will deduce that the result is another `Int`.

2 Pairs

For the rest of this paper we will investigate how to combine primitive data types in an arbitrary way. Our primitive data types will have lowercase names like `x` and `y`.

If `x` and `y` are primitive types, then $\texttt{x} \cdot \texttt{y}$ is a type of ordered pairs of an `x` and a `y`. This is called the *product type* of `x` and `y`. For instance the type $\texttt{Int} \cdot \texttt{Char}$, the product type of `Int` and `Char`, represents pairs of integers and characters. Values which satisfy this type include `(4, a)` and `(2, x)`

Note that this is an ordered pair, so in $\texttt{Int} \cdot \texttt{Int}$ the values `(3,4)` and `(4,3)` are distinct.

3 Option type

There is another way to combine types, which is summing them to create an *option type*. To explain this clearly we can think of creating a product type $\texttt{x} \cdot \texttt{y}$ as a two-step process:

1) Create a pair of blanks `(_,_)`. The left blank is associated with `x` and the right with `y` .

2) Fill in all the blanks with values of their associated type.

For sum type, the first step is the same, but the second is changed to

2) Choose *one* blank and fill it with a value of its associated type.

For instance Int + Char can be inhabited by (5, _) and (_, 'a'). This explains why it's called an option type - intuitively, a value of that type has an option of being any one of its subtypes. However, note that it is an ordered choice, so for example the type `Int + Int` is not the same as `Int` because `(_,5)` is not the same as `(5,_)`.

4 Counting types

Why do we define product and option types in this strange way? One reason is that sizes of sum and product types are the sums and products of the sizes of respective subtypes - if `#(x)` denotes the number of types that inhabit `x`, then `#(x+y) = #(x) + #(y)`, and likewise for product types

5 Identities

The reason why we call these Algebraic Data Types is because we can add and multiply them. It's also nice to notice that multiplication and addition are both associative and commutative - for instance, `(Int+Char)+Int` is inhabited by `((1,a),3` which corresponds in `(Int, (Char, Int))` to `(1,(a,3))`, and so the two types are equal. Actually, we really mean isomorphic, but from now on equality shall mean isomorphic. This also allows us to define `(x+y+z)` in a natural way.

Once we have these we really should begin thinking of identities. There should be a type `0` such that `0 + x = x` and a type `1` such that $1 \cdot x = x$. Consider that `#(1) = 1` and `#(0) = 0`, they should be types that are only inhabited by one value, and cannot be inhabited by any value at all, respectively. You should think of `1` as the type that represents the empty set - you can't say that two empty sets are different. `0` is commonly known as bottom.

The proof of the identities is quite intuitive - `0 + x` is eiither a `0` or an `x`, but by definition it can't be a `0` so it must be an `x`. $1 \cdot x$ is a pair of a `1` and a `x`, but the first element has only one possibility (ie the empty set), so the only freedom lies in the second blank.

Another identity that follows is $0 \cdot x = 0$ - you can't satisfy the first blank, but you must satisfy both blanks to satisfy the whole type, thus you can't satisfy the whole type.

6 Finite types

Consider the type `1 + 1` which we shall call `2`. It is inhabited by two values, `({}, _)` and `(_,{})`. In other words it is isomorphic to what we normally call the boolean type. In a similar way we can define types `3`, `4`, etc, which are called the finite types because they represent an enumeration of a finite number of choices.

Using this, we can simplify the type `x + x` to be `2x`, because if `t` satisfies `x` we can map `(t, _)` to `("left", t)` and `(_, t)` to `("right", t)`.

The distributive law also holds for types - $a \cdot (x + y) = ax + ay$

7 Lists

How shall we define a lists of things as an ADT? We can do so recursively. A list is either an empty list or a non-empty list; if it is non-empty, it is equal to a pair where the first element is the head of the list and the second element is another list. Hence, if `L[x]` represents a list of `x`s,

$L[x] = 1 + x \cdot L[x]$

where a list of `x`s is written as a sum type of either an empty list (`1`) or a pair of an `x` and a `L[x]`. This is a recursively definition because `L[x]` is defined in terms of itself. We can clean it up a bit and "manipulate" this equation algebraicly

$$
\begin{aligned}
L &= 1 + xL \\
L(1-x) &= 1 \\
L &= \frac{1}{1-x} \\
&= 1 + x + x^2 + x^3 + ...
\end{aligned}
$$

which says that a list is either an empty list, or a pair, or a triple, or...