What even *are* types?

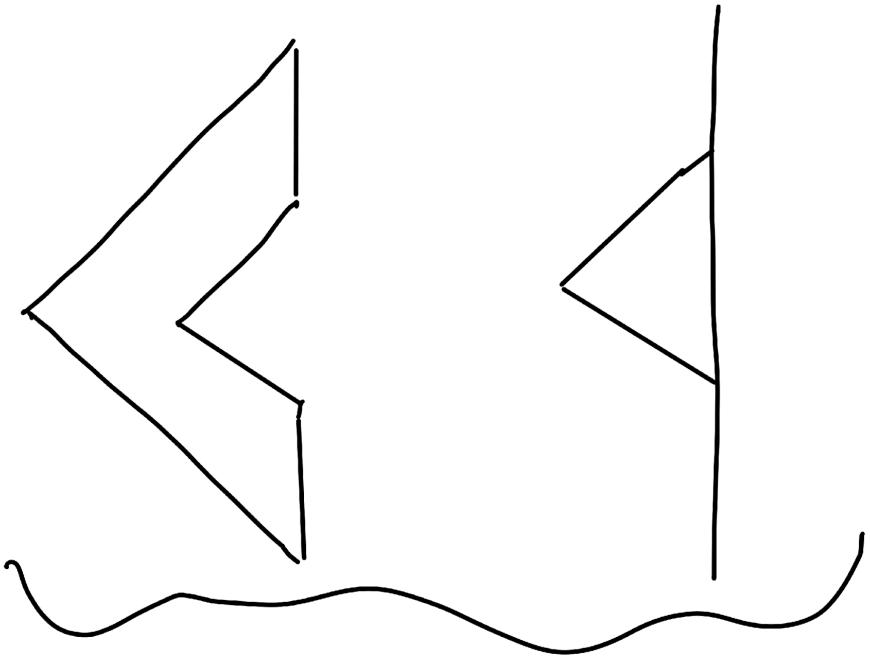
Clarissa Littler

You may have heard the phrases "statically typed" and "dynamically typed" to describe programming languages: static typing means that a language has a phase where the types of all the data and functions are checked to make sure the program makes sense, and dynamic typing means that it does not. I think it's better to call "statically typed" languages just *typed* and "dynamically typed" languages *untyped* since types that can't get checked or understood by the compiler/interpreter can't really **do** anything except crash the program when a type error happens.

So what the heck even are types? They're things that label and describe data! Most typed languages have types for things like integers, strings, booleans, or arrays. Have you ever noticed that Scratch is a typed language? It's a really simple type system but it's there! The shapes of blocks correspond to the types. Triangular edges are booleans. Round edges are numbers or characters. Scratch won't let you put the wrong shape in a spot, which is its version of type checking.

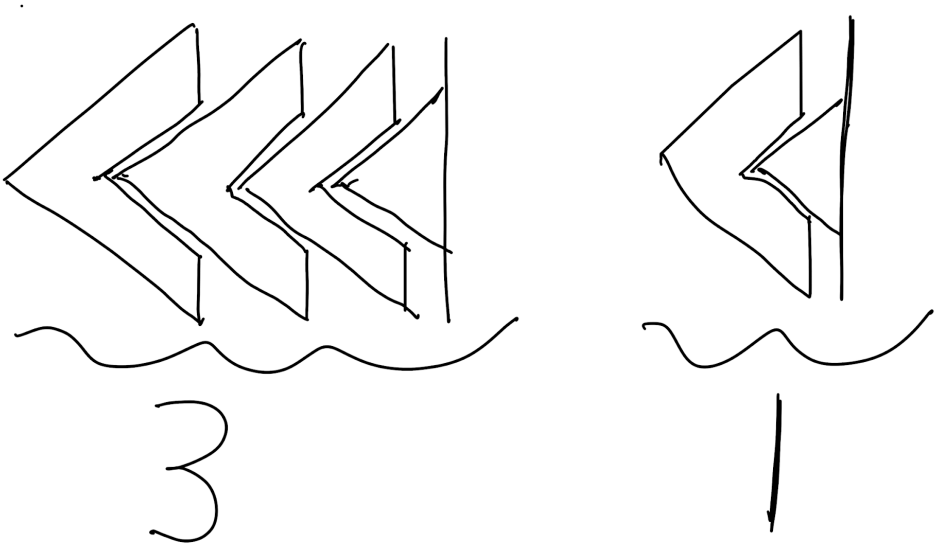
This got me thinking: shapes are a really good way to describe types! In particular, they're a good way to start talking about one of the cooler kinds of types that languages like Idris, also featured in this issue, have: *inductive* types.

Inductive types are a language feature that lets a programmer define an entirely new type *by how it's built*. Here's our first inductive type as a scheme of shapes:



These two connectors form the entire description of this type. You can make any data of this type by fitting them together along their connectors.

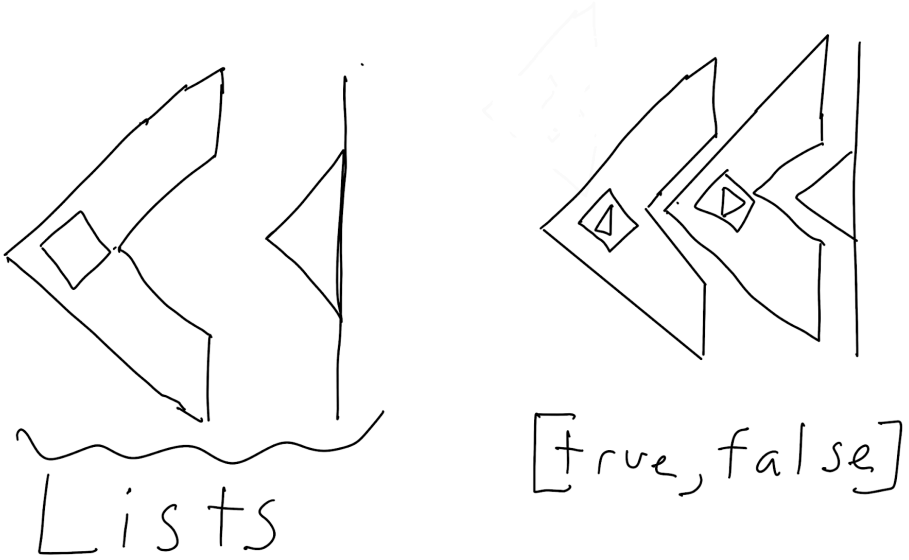
What *is* this data, though? In a very real sense, these are the natural—that is, whole and greater or equal to zero—numbers. The end-piece represents "0" and you add one to the number for each triangular piece there is. Like so



Uhhh why on earth would you represent numbers this way? The neat thing about writing numbers this way is that if I hand you one of these numbers, `n`, and ask you to do something `n` many times—no matter what it is—you can easily write an algorithm that describes exactly how to do it because *every* traversal, or movement across the data, has the exact same logic as counting the data. For example, let's assume "the thing to do" is "jumping jacks" because exercise is important!

> Look at the front piece of the data. If it's an end piece, stop exercising. If it's a triangle piece, peel it off, do a jumping jack, and then repeat this process for the rest of the data.

Now, let's modify our drawing a little bit so that it holds a slot for some other kind of data inside it: a left or right facing triangle, which represents *true* and *false* respectively.



Now we've got a thing that can hold data and ways to build it. It looks *almost* the same but now instead of numbers we've defined lists! Lists of booleans to be specific!

You can write your jumping-jack traversal for these lists like

> Read the front of the list. If it's an end piece, stop exercising. If it's not, look at the data inside it: if it's true then jump your jack, if it's false don't. Either way, repeat this process with the rest of the list.

Let's really put these tricks to the test. We're going to define a new datatype that's super-important in storing data for efficient search in databases.

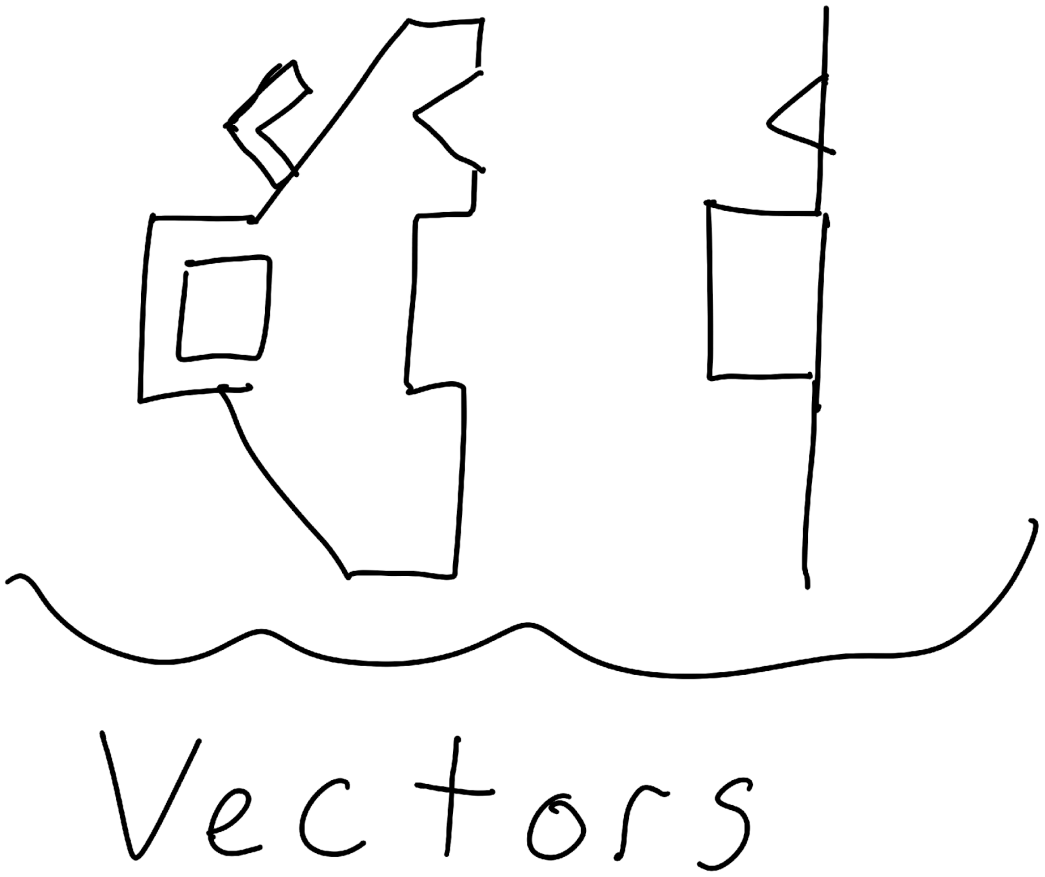
Now how do we describe the generic traversal? Take a second and see if you can figure it out before

I spoil it. Hint: there are two possible answers depending on a choice you make.

Were you able to get it? When you want to compute something for each part of the datastructure you either start at a branching part or an end node. If you reach an end node you're done with that part of the computation. If you have a branch, you need to act on the piece of data stored in the branch and then handle the left tree and then the right tree. (Or the right tree and then the left tree. You actually have a choice here!)

There's such regularity to these schemes for computation that languages like Idris and Agda can give you skeletons of code to fill in by just looking at the possible shapes the data can have.

There's even more complicated shapes you can make, which depend on the size of other shapes. Here's a drawing of the vectors from our Idris article



They have both the tabs for the vectors that fit into each other *and* tabs to count the length of the list up near the top. You can see that as you put them together you're also building up the list as a natural number shape like we defined earlier! If you check the Idris article this corresponds *exactly* to the dependency the vector type has on its length.

The punchline here is that types have a structure that can be used to do neat things that make programming easier and let us express complex ideas simply. So as you learn more about languages like Idris, whenever you're confused just go ahead and try drawing the shape of the types!

Happy hacking!