A question that many students have on first seeing a recursive type, such as the celebrated type \( \text{rec } t \rightarrow t \) of “untyped” \( \lambda \)-terms, is why is it not an inductive type? After all, in their programming experience, inductive types such as trees and lists and whatnot are represented as recursive types. Moreover, this representation is so common, it is natural to consider that recursive types just \textit{are} inductive types. So what is the difference?

The technically correct, but uninformative, answer is that the type operator \( t.t \rightarrow t \) is not positive, because the occurrence of \( t \) on the left of an arrow is precluded as part of the definition of positivity. The left side of the arrow as “negative”, and the right side as “positive”, so \( t \) occurs both positively and negatively in \( t \rightarrow t \), and that’s the rub.\(^1\) Consequently, neither \( \mu(t.t \rightarrow t) \) nor \( \nu(t.t \rightarrow t) \) are well-formed, but the recursive type, \( \text{rec } t \rightarrow t \), is, and that’s that.

But this begs the question, why are inductive (and coinductive) types defined from only positive type operators, and not more generally? A very informal way to justify it is that it is possible to \textit{draw} on paper the elements of an inductive type using the familiar box-and-pointer diagrams. This fact is directly related to the positivity requirement, because it ensures that the values of the type are built up in well-defined “stages” such that pointers always refer to a value from an earlier stage.\(^2\)

To see how this works, let us consider the most fundamental inductive type of all, the set \( \mathbb{N} \) of natural numbers. We can think of the natural numbers in terms of the following development:

\[
\emptyset \subseteq \{0\} \subseteq \{0, s(0)\} \subseteq \{0, s(0), s(s(0))\} \subseteq \cdots \subseteq \mathbb{N}
\]

The set \( \mathbb{N} \) is the \textit{colimit} of the approximations that lead up to it. At each stage we take what we have so far and close it up under zero and the successor operation to obtain a better approximation to the colimit. The elements of the co-limit are those that arise at some stage of approximation—the colimit is their union.

Correspondingly, the type of natural numbers is the inductive type \( \mu(t. \text{unit } + t) \), provided that \( \text{fold}()(-) \) and the injections into the sum are interpreted eagerly.\(^3\) The approximants of \( \text{nat} \) are determined by successively unrolling the inductive type, starting with the empty type:\(^4\)

\[
\text{void } \subseteq \text{unit } + \text{void } \subseteq \text{unit } + \text{unit } + \text{void } \subseteq \cdots \subseteq \text{nat}.
\]

\(^1\)It might be more intuitive to say that the left side of a function type is an “input” position, and the right side an “output” position, in which case the requirement for inductive (and coinductive) types is that the type operator be an “output operator”, rather than an “input operator.”

\(^2\)In other words the pointers act like successors, which induces an induction principle based on these stages.

\(^3\)Otherwise the type is polluted by all manner of extraneous elements, including \( \omega \), which its own successor.

\(^4\)Here we use the subset notation metaphorically to indicate that there is a canonical coercion from the smaller to the larger type.
Thinking of the elements at each stage, we have, starting from the second stage in the above sequence,
\[
\text{fold}(l \cdot \langle \rangle), \text{fold}(r \cdot \text{fold}(l \cdot \langle \rangle)), \text{fold}(r \cdot \text{fold}(r \cdot \text{fold}(l \cdot \langle \rangle))), \ldots
\]
And these, of course, correspond to the natural numbers, taking fold to be a “pointer” to a datum tagged either 1 or r according to whether it is zero or the successor of another such value. Notice that the pointer so-called points to the predecessor of a non-zero natural number.

A similar account can be given for coinductive types, only backwards.\(^5\) As a first approximation, \textit{everything} is a candidate value of a coinductive type, dually to the starting point for inductive types. At each round we knock out those candidates that are not either zero or the successor of an earlier candidate. So at round one we kill everything but zero and the successor of any old junk, including zero. At round two we kill off everything but zero, the successor of zero, and the successor of the successor of anything at all. The type of co-natural numbers is the \textit{limit} of these approximations, their intersection.\(^6\) Notice that \(\omega\), the infinite composition of successors, survives into the limit stage! Here we depend on laziness so that the successors are not all computed at once, but only revealed as they are required.

The main point is that the idea of taking the colimit, or limit, of a sequence of approximants applies \textit{only} when the type operator in question is positive. Otherwise it is impossible to think in terms of stages, at least not in the immediately evident sense we are considering here.\(^7\) Why is that? Let’s consider the recursive type \(D \triangleq \text{rec t is } t \rightarrow t\), which does not satisfy the positivity requirement, and see what happens if we attempt to interpret it as arising as the colimit of its approximants as we did for inductive types:

\[
D_0 \triangleq \text{void} \subseteq^? D_1 \triangleq D_0 \rightarrow D_0 \subseteq^? D_1 \rightarrow D_1 \subseteq^? \cdots \subseteq^? D
\]

By definition \(D_0\) has no elements; every element of this type survives into the next, and subsequent, rounds. There are infinitely many partial functions of type \(D_0 \rightarrow D_0\), any partial function at all, including the totally undefined function, precisely because the domain type is empty. But do these survive in the next round as elements of \(D_1 \rightarrow D_1\)? Not at all! The totally undefined function survives, as does the identity function, but other functions inhabiting \(D_1\) only because the domain is empty are ruled out as elements of \(D_2\), exactly because \(D_1\) is not empty! Thus the sequence of types \(D_0, D_1, D_2, \ldots\) oscillates wildly, with some elements surviving into successive stages, and others being killed off. The sequence is not obviously heading anywhere, and so it should be plausible that it has no colimit. The situation is no better if we attempt to interpret the recursive type coinductively, for once again the sequence of iterates does not converge in any obvious sense, and so no limit can be expected.

Positive type operators have well-defined colimits and limits of their iterates, but general type operators do not. Thus recursive types cannot, in general, be interpreted as inductive or coinductive types. In fact recursive types greatly extend the expressive power of a language beyond what is possible with mere inductive and coinductive types. For a start, they introduce undefinedness (partiality). This may seem more like a bug than a feature, but it is in fact fundamental: \textit{rect is...}  

\(^5\)They are Ginger Rogers to the inductive Fred Astaire.  
\(^6\)The limit here is sometimes called the \textit{inverse limit}, in which case the colimit is called the \textit{direct limit}. If the terminology feels sort of backwards, it’s because two traditions collided and came out opposite one another.  
\(^7\)Dana Scott invented a much more sophisticated idea of staging that applies to recursive types, called the \textit{inverse limit construction}, which we shall not detail here.
$t \to t$ makes no sense if the function type is \emph{total},$^8$ but makes perfect sense if it is \emph{partial}. The entire theory of recursive types, and of recursive self-reference in general, relies on this observation.

**References**


$^8$For reasons of cardinality.