Example

Let's assume we would like to synthesize a function f with the following properties:

- $f : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$
- $f\ 2\ 1 \equiv 3$
- Identity
  - $\prod_{(x:\mathbb{N})} f\ x\ 0 \equiv x$ (Type Theoretically)
  - $\forall x \in \mathbb{N}.\ f\ x\ 0 \equiv x$ (Classically)

One way we might synthesize this such that the properties always hold could go something like this:

Step 1 - Setup

We start with a simple definition with a single hole representing the entire function to be synthesized:

$$\begin{array}{l} f : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ f = \boxed{\mathbb{N} \to \mathbb{N} \to \mathbb{N}}_h \end{array} \quad \text{with} \quad \begin{array}{l} \texttt{ex} : h\ 2\ 1 \equiv 3 \\ \texttt{id} : \prod_{(x:\mathbb{N})} h\ x\ 0 \equiv x \end{array}$$

Here h denotes the hole that we need to fill to complete the synthesis problem. The properties from above are transformed to be over the hole instead of f at this step.

Step 2 - Introduction of Arguments

In the first step we bring both arguments to h into scope by introducing them on the LHS of the equation. This also alters the properties:

$$\begin{array}{l} f : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ f\ x\ y = \boxed{\mathbb{N}}_h \end{array} \quad \text{with} \quad \begin{array}{l} \texttt{ex} : (x \equiv 2) \wedge (y \equiv 1) \to (h \equiv 3) \\ \texttt{id} : (y \equiv 0) \to (h \equiv x) \end{array}$$

Step 3 - Splitting on y

In the third step of synthesis, we split on the argument y. This yields two new clauses from the structure of $\mathbb{N}$.

$$\begin{array}{l} f : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ f\ x\ 0 \qquad\quad = \boxed{\mathbb{N}}_{h_1} \\ f\ x\ \underbrace{(S\ y')}_{y} \ = \boxed{\mathbb{N}}_{h_2} \end{array} \quad \text{with} \quad \begin{array}{ll} h_1 & \boxed{\begin{array}{l} \texttt{ex} : (x \equiv 2) \wedge (\underbrace{0}_{y} \equiv 1) \to (h_1 \equiv 3) \\ \texttt{id} : (\underbrace{0}_{y} \equiv 0) \to (h_1 \equiv x) \end{array}} \\ h_2 & \boxed{\begin{array}{l} \texttt{ex} : (x \equiv 2) \wedge (\underbrace{(S\ y')}_{y} \equiv 1) \to (h_2 \equiv 3) \\ \texttt{id} : (\underbrace{(S\ y')}_{y} \equiv 0) \to (h_2 \equiv x) \end{array}} \end{array}$$

We've now split this into two synthesis problems. We may also reduce the properties for the h_1 and h_2 in a few ways. First we may remove `ex` from h_1 because one of it's premises is a contradiction($0 \equiv 1$). From h_2 we can remove `id` because $((Sy') \equiv 0)$ is also a contradiction.

This give the new configuration:

$$
\begin{array}{lll}
f : \mathbb{N} \to \mathbb{N} \to \mathbb{N} & & h_1 \quad \boxed{\texttt{id} : (0 \equiv 0) \to (h_1 \equiv x)} \\
f\ x\ 0 & = \boxed{\mathbb{N}}_{h_1} \quad \text{with} & \\
f\ x\ (S\ y') & = \boxed{\mathbb{N}}_{h_2} & h_2 \quad \boxed{\texttt{ex} : (x \equiv 2) \wedge ((S\ y') \equiv 1) \to (h_2 \equiv 3)}
\end{array}
$$

We may also make a few simplifications to both properties. For `id` in h_1 we may conclude $h_1 \equiv x$ because $(0 \equiv 0)$ is true. For h_2 we can take advantage of the congruence of S in $((S\ y') \equiv 1)$ to get $y' \equiv 0$. This gives us the configuration:

$$
\begin{array}{lll}
f : \mathbb{N} \to \mathbb{N} \to \mathbb{N} & & h_1 \quad \boxed{\texttt{id} : (h_1 \equiv x)} \\
f\ x\ 0 & = \boxed{\mathbb{N}}_{h_1} \quad \text{with} & \\
f\ x\ (S\ y') & = \boxed{\mathbb{N}}_{h_2} & h_2 \quad \boxed{\texttt{ex} : (x \equiv 2) \wedge (y' \equiv 0) \to (h_2 \equiv 3)}
\end{array}
$$

Now we will begin to solve each of the remaining holes.

Step 4 - Solving the Base Case

We have an easy solution for h_1 because we have a constraint that tells us what the hole *must* be filled with.

$$
\begin{array}{lll}
f : \mathbb{N} \to \mathbb{N} \to \mathbb{N} & & \\
f\ x\ 0 & = x \quad \text{with} & h_2 \quad \boxed{\texttt{ex} : (x \equiv 2) \wedge (y' \equiv 0) \to (h_2 \equiv 3)} \\
f\ x\ (S\ y') & = \boxed{\mathbb{N}}_{h_2} &
\end{array}
$$

Step 5 - Filling with a Constructor

We can make some progress on h_2 (now h), by filling the hole with the constructor $(S\ \boxed{\mathbb{N}})$.

$$
\begin{array}{lll}
f : \mathbb{N} \to \mathbb{N} \to \mathbb{N} & & \\
f\ x\ 0 & = x \quad \text{with} & h \quad \boxed{\texttt{ex} : (x \equiv 2) \wedge (y' \equiv 0) \to ((S\ h) \equiv 3)} \\
f\ x\ (S\ y') & = S\ \boxed{\mathbb{N}}_{h} &
\end{array}
$$

This again will simplify over the congruence of S and give:

$$
\begin{array}{lll}
f : \mathbb{N} \to \mathbb{N} \to \mathbb{N} & & \\
f\ x\ 0 & = x \quad \text{with} & h \quad \boxed{\texttt{ex} : (x \equiv 2) \wedge (y' \equiv 0) \to (h \equiv 2)} \\
f\ x\ (S\ y') & = S\ \boxed{\mathbb{N}}_{h} &
\end{array}
$$

Step 6 - A Recursive Call to f

In this step we will fill the hole with a recursive call.

$$
\begin{array}{lll}
f : \mathbb{N} \to \mathbb{N} \to \mathbb{N} & & \\
f\ x\ 0 & = x \quad \text{with} & h_1, h_2 \quad \boxed{\texttt{ex} : (x \equiv 2) \wedge (y' \equiv 0) \to ((f\ h_1\ h_2) \equiv 2)} \\
f\ x\ (S\ y') & = (S\ (f\ \boxed{\mathbb{N}}_{h_1}\ \boxed{\mathbb{N}}_{h_2})) &
\end{array}
$$

Step 7 - Using Our Properties

To wrap up the problem, we take advantage of one of the properties of out program, the identity property. Apply the property for $x \equiv 2$, we get $f\ 2\ 0 \equiv 2$. Which is the form of `ex` when h_1 and h_2 are filled with 2 and 0 respectively. We get 2 and 0 from their being bound to x and y'. This gives us the satisfaction of `ex` and a solution for the problem.

$$
\begin{array}{ll}
f : \mathbb{N} \to \mathbb{N} \to \mathbb{N} & \\
f\ x\ 0 & = x \\
f\ x\ (S\ y') & = (S\ (f\ x\ y'))
\end{array}
$$

Reflection

In this example we do not treat the properties as synthesis problems in their own right. Ideally, we would synthesize their proof terms as we went! I've begun calling this process Co-Synthesis(to further confuse and frustrate the category theorists) because it involves synthesizing programs *along with* their proofs at the same time. They inform eachothers' synthesis processes. There is also a type-theoretic interpretation to this process which encodes each recursive step as the synthesis of a dependent-sum type. More on this soon!

What this example doesn't deal well with is *how* the synthesizer would know to move along a good path. The direct process taken here is almost certainly not how the synthesizer would proceed. There would be many dead ends(potentially infinitely deep ones) that must be avoided in one way or another.

The other thing that was a bit "hand-wavy" is the way we use properties and propositions in synthesis. This involves a non-trivial check to see if any propositions match the pattern of a hole. (This was the motivation for some of my thinking in the program search/code reuse space).

Going forward I think there are a set of general principles and inference rules on Dependent-Sum types that can describe the process followed above. Writing up the Sum-Type version of this example will be the first step towards these rules.