The slip data structure introduces the name \textit{slip} with the following axioms:
\begin{align*}
\text{slip} &= [X; \text{slip}] \\
B &= [X; B] \Rightarrow B: \text{slip}
\end{align*}
where $X$ is some given type. Can you implement it?

That second axiom is not induction; it is coinduction, defining \textit{slip} to be the largest solution of the construction axiom. (If it were induction, the two axioms would define \textit{slip} to be \textit{null}.) If lists and recursive definition are implemented, as they are in some “lazy functional” languages like LazyML and Haskell, then \textit{slip} is already implemented by the first axiom. It's strange because the recursion doesn't seem to have a base, so \textit{slip} is an infinite structure:
\begin{align*}
\text{slip} &= [X; [X; [X; [...] ] ] ]
\end{align*}
In C we have to use pointers.
\begin{verbatim}
struct slip {X left; slip *right;};
\end{verbatim}
Although recursive data types are seldom implemented, recursive functions usually are implemented. (This is a strange inconsistency in the design of programming languages; the reasons for recursion and the implementation of recursion are exactly the same for data types as for functions and procedures.) We can define
\begin{align*}
\text{slip} &= \{ X \mid 0 \rightarrow X \} | 1 \rightarrow \text{slip}
\end{align*}
or
\begin{verbatim}
slip = \langle n : 0,1. \text{ if } n=0 \text{ then } X \text{ else } \text{slip} \rangle
\end{verbatim}
This function definition will be a problem in a language that wants you to state the result type. The number of further arguments depends on the values of previous arguments.