To understand *complete induction* on causal order, it helps to review ordinary (simple) induction on numbers, $\mathbb{N}$, and compare it to *complete induction on numbers*. These induction forms are instances of primitive recursion used to define addition and subtraction in Kleene's formalism. Let's review primitive recursion first.

1 Primitive Recursion

informal notation
$f(0, y) = g(y)$
$f(x + 1, y) = h(x, y, f(x, y))$

Note, the order of arguments to h does not matter, another good choice seen in textbooks is $h(x, f(x, y), y)$.

using if-then-else
$f(x, y) =$ `if` $x = 0$ `then` $g(y)$ `else` $h(x - 1, y, f(x - 1, y))$

Note, since we don't use $x + 1$ as an input form, we need $x - 1$ as an operation.

Using *currying* of arguments
$f(0)(y) = g(y)$
$f(x + 1)(y) = h(x)(y)(f(x, y))$

Using *functions as outputs*
$f(0) = \lambda(y.g(y))$
$f(x + 1) = \lambda(y.h(x)(y)(f(x)(y)))$

f as a function and if-then-else
$f = \lambda\big(x.\lambda\big(y.$ `if` $x = 0$ `then` $g(y)$
`else` $h(x)(y)(f(x - 1)(y))\big)\big)$

The `fix` format is the most general form of recursion.

* `fix`$\big(\lambda\big(f.\lambda(x.\lambda(y.$ `if` $x = 0$ `then` $g(y)$ `else` $h(x)(y)(f(x - 1)(y)))))\big)$

The fix-combinator treats f recursively. The first step of reduction yields the result

$\lambda(x.\lambda(y.$ `if` $x = 0$ `then` $g(y)$ `else` $h(x)(y)($`fix`$(—)(x - 1)(y))))$

where `fix`(—) is the *entire expression* labeled *.

This shows us the *essence of recursion*, we reuse the entire definition during evaluation. We can define all recursive functions this way. But some do not terminate, e.g. `fix`$(\lambda(x.x))$ reduces in one step to itself! We call this special term `div` or sometimes $\perp$ (the $\perp$ form is called Scott's bottom element.)

The induction rule as primitive recursion.
Simple induction is the rule (in curried form):
$P(0) \Rightarrow (\forall x.(P(x) \Rightarrow P(x+1))) \Rightarrow \forall x.P(x)$

We can derive a realizer for it using `fix`. We know the form of the realizer is
$\lambda\big(b.\lambda\big(h.\lambda(x.\ \texttt{if}\ x = 0\ \texttt{then}\ b\ \texttt{else}\ h(x-1)(f(x-1)))\big)\big)$

Where $f(x)$ should prove $P(x)$; so that means that f should be the function we want in $\forall x.P(x)$!
So $f = \lambda(b.\lambda(h.\lambda(x,\ \texttt{if}\ x = 0\ \texttt{then}\ b\ \texttt{else}\ h(x-1)f(x-1))))$.

If we supply f with a specific proof b of $P(0)$ and a specific function h_0 for $\forall x.(P(x) \Rightarrow P(x+1))$, then we get
$f(b_0)h_0) = \lambda(x.\ \texttt{if}\ x = 0\ \texttt{then}\ b_0\ \texttt{else}\ h_0(x-1)(f(b_0)(h_0)(x-1)))$
which is the evidence of $\forall x.P(x)$.

This recursive function is given exactly by
$\texttt{fix}\ \big(\lambda\big(f.\lambda(b.\lambda(h.\lambda(x.\ \texttt{if}\ x = 0\ \texttt{then}\ b$
$\texttt{else}\ h(x-1)(f(b)(h)(x-1))))))\big)$

It is the realizer for simple induction. We can see this intuitively, and we can take it as evidence for the induction axiom, atomic evidence.

One step of reduction reduces this to
$\lambda\big(b.\lambda\big(h.\lambda(x.\ \texttt{if}\ x = 0\ \texttt{then}\ b$
$\texttt{else}\ h(x-1)\ \texttt{fix}(-)(b)(h)(x-1))\big)\big)$

If we supply b_0 and h_0 we get
$\lambda(x.\ \texttt{if}\ x = 0\ \texttt{then}\ b_0\ \texttt{else}\ h_0(x-1)\ \texttt{fix}(-)(b_0)(h_0)(x-1))$
as the realizer for $\forall x.P(x)$ provided
b_0 proves (is evidence for) $P(0)$ and
h_0 proves (is evidence for) $\forall x.(P(x) \Rightarrow P(x+1))$

We can see the rule term for induction in Nuprl as a *primitive constant* for this term
$\texttt{ind}(x; b_0; u, f.h_0(u)(f(u)))$.

The constant is designed to be a good logical evidence form. It is a *primitive induction constant*, in logical form.

We now want to find the realizer for complete induction on $\mathbb{N}$. Here is the logical form of the rule.

Complete induction
$\forall x.(\forall y.(y < x \Rightarrow P(y)) \Rightarrow P(x)) \Rightarrow \forall x.P(x)$

It turns out to be handy to write the hypothesis in the following equivalent way (by "un-currying"):
$\forall y : \{z : \mathbb{N} | z < x\}.P(y)$

We mean by $\{z : \mathbb{N} | z < x\}$ the type of all elements of $\mathbb{N}$ for which we have evidence that $z < x$, but we will not use this evidence in computations. We call this a *set type* or a *refinement type*. We can use this as a sort in FOL.

The *proof rule for complete induction* has the form

$$\begin{aligned}
n : \mathbb{N} &\vdash P(n) \text{ by } \mathtt{cind}(n; x, f.__) \\
n : \mathbb{N}, x : \mathbb{N}, f : \forall y : \{z : \mathbb{N} | z < x\} &\vdash P(x) \text{ by } \mathtt{ap}(f; \mathtt{exp}_1(x); u_1.__) \\
&\vdash \mathtt{exp}_1(x) \in \{z : \mathbb{N} | z < x\} \\
u_1 : P(\mathtt{exp}_1(x)) &\vdash P(x) \text{ by } \mathtt{ap}(f; \mathtt{exp}_2(x); u_2.__) \\
&\vdots \\
u_1 : P(\mathtt{exp}_1(x)), \ldots, u_k : P(\mathtt{exp}_k(x)) &\vdash P(x) \text{ by } \mathtt{ap}(f;\ \mathtt{exp}_k(x); u_k.__) \\
&\vdash\ \mathtt{exp}_k(x) \in \{z : \mathbb{N} | z < x\} \\
&\vdash P(x) \text{ by } g(x, u_1, \ldots, u_k)
\end{aligned}$$

The evidence term is $\mathtt{cind}(n; x, f.g(x, u_1, \ldots, u_k))$.

How does it compute?

How do we compute with $\mathtt{cind}(n; x, f.g(x, u_1, \ldots, u_k))$?

The reduction rule is that in one step we reach an expansion of $\mathtt{cind}(n; x, f.g(x, f(\mathtt{exp}_1(x)), \ldots, f(\mathtt{exp}_k(x))))$ namely, substituting $\mathtt{cind}(\mathtt{exp}_i(n); x, f.\_\_)$ for $f(\mathtt{exp}_i(x))$ to get
$g(n, \mathtt{cind}(\mathtt{exp}_1(n); \_\_), \ldots, \mathtt{cint}(\mathtt{exp}_k(n), \_\_))$

This is like defining
$f(x) = g(x, f(\mathtt{exp}_1(x)), \ldots, f(\mathtt{exp}_k(x))$ hence
$f = \lambda(x, g(x, f(\mathtt{exp}_1(x)), \ldots, f(\mathtt{exp}_k(x))))$

and finally
$\mathtt{fix}(\lambda(f.\lambda(x.g(x, f(\mathtt{exp}_1(x)), \ldots, f(\mathtt{exp}_k(x))))))$

It is possible to derive this form from the one for simple (primitive) induction and conversely.

Extra credit exercise. Prove complete induction from primitive induction and conversely. Use the form of the realizers to guide your proofs. This is not easy.