Recall the issue we faced last time:

What is the realising computational evidence for

\[ \text{Induction: } (A(0) \land \forall x. A(x) \Rightarrow A(sx)) \Rightarrow \forall x. A(x) \] ?

How do we use \( A(0) \) and the function witnesses

\( \forall x. (A(x) \Rightarrow A(sx)) \) to produce the function for \( \forall x. A(x) \)?

We need \( \lambda (h. \ \text{spread}(h, b, f, \_)) \) and?

must provide a realiser for \( \forall x. A(x) \).

We saw that \( A(0) \) \( A(1) \) \( A(2) \) \ldots

by \( \ldots \)

by \( \ldots \)

by \( \ldots \)

\[ \begin{align*}
& \quad a_0 \quad a_1 \\
& \quad f(0)(a_0) \quad f(1)(f(0)(a_0)) \quad f(2)(f(0)(a_0))
\end{align*} \]

But \( \_ \)? can't be \ldots. It must be a computation expression, a program. Something like \( \text{let rec ind}(x) = f(x-1, \text{ind}(x-1)) \) might work. In this case we need \( x > 0 \) to allow \( f(x-1) \).

What about the \( 0 \) case? Then we use the base, \( b \). Thus

\[ \text{ind}(x) = \begin{cases} 
  b & \text{if } x = 0 \\
  f(x-1, \text{ind}(x-1)) & \text{else}
\end{cases} \]

In the Nuprl style proof rules we use

\[ \text{Ind}(x; b; m, i. f(m, i)) \] with these computation rules.

\[ \text{Ind}(0; b; \_; b) \quad \text{Ind}(s0; b; m, i. f(m, i); f(x, \text{ind}(x; b; m, i. f(m, i)))) \]
We use the induction form for all computation on numbers. For example, to decide whether a number is zero, we compute with $\text{ind}(x; *; m, i, i)$ where $i$ is any computation form that "shuts" like $\text{ap}(0, 0)$.

We can use $\text{ind}$ to prove statements such as

1. $\forall x. (Z(x) \lor \neg Z(x))$

2. $\forall x, y. (\text{Eq}(x, y) \lor \neg \text{Eq}(x, y))$

3. $\forall x (\neg Z(x) \Rightarrow \exists y. \text{Suc}(y, x))$

We can use induction to prove

4. $\forall x, y. \exists z. \text{Add}(x, y, z)$

5. $\forall x, y. \exists z. \text{Mult}(x, y, z)$

We will prove 4 below. You should try 1, 2, 3, 5 as exercises. Also try 6 below.

6. Define $x < y$ iff $\exists z. (x + z = y \land z \neq 0)$. Show:
   
   (a) $x < s(x)$
   (b) $(x < y \land y < z) \Rightarrow x < z$
   (c) $\neg (x < x)$
1. ∀y. Add(0, y, 3) \iff Kleene Ax & add(0, y) = y
2. ∀x, y, z. (Add(x, y, z) \Rightarrow Add(s(x), y, s(z)))

Kleene Ax 19 add(s(x), y) = s(add(x, y))

We show Theorem ∀x, y, z. Add(x, y, z) by induction (Kleene Ax 19)

\[ \vdash \exists z. Add(x, y, z) \text{ by } \lambda(x, y, z) \]
\[ x : \mathbb{D} \vdash \exists z. Add(x, y, z) \text{ by } \lambda(y, z) \]
\[ x : \mathbb{D}, y : \mathbb{D} \vdash \exists z. Add(x, y, z) \text{ by } \lambda(x, y, z) \]
\[ \vdash \exists z. Add(0, y, z) \text{ by } \lambda(y, z) \]
\[ \vdash 0 \text{ by } y \]
\[ \vdash Add(0, y, z) \text{ by } \lambda(y, z) \]
\[ \vdash 0 \vdash \exists z. Add(s(0), y, z) \text{ by } \lambda(y, z) \]
\[ x : \mathbb{D}, y : \mathbb{D}, z : \mathbb{D} \vdash \exists z. Add(x, y, z) \text{ by } \lambda(x, y, z) \]
\[ x : \mathbb{D}, y : \mathbb{D}, z : \mathbb{D} \vdash \exists z. Add(s(x), y, z) \text{ by } \lambda(x, y, z) \]
\[ \vdash s(0) \text{ by } y \]
\[ \vdash Add(s(0), y, z) \text{ by } \lambda(y, z) \]
\[ \vdash Add(5(0), y, z) \text{ by } \lambda(y, z) \]
\[ \vdash \lambda(x, y, z). Add(x, y, z) \Rightarrow Add(s(x), y, s(z)) \text{ by } \lambda(x, y, z) \]
\[ \vdash \lambda(x, y, z). Add(x, y, z) \Rightarrow Add(s(x), y, s(z)) \text{ by } \lambda(x, y, z) \]
\[ \vdash i, s(0), a \vdash Add(s(0), y, s(z)) \text{ by } \lambda(y, z) \]
The computation form \( \text{ind}(x; b; u, i, f) \) is recursive. You might know that recursion is more expensive than iteration. Can we use computational forms such as

\[
\text{for } i = 0 \text{ to } n \text{ do } x := f(i, x) \text{ od}
\]

as forms of induction? We'd be computing

\[
f(0, x_0), f(1, f(0, x_0)), f(2, f(1, f(0, x_0))), \ldots
\]

If we set \( x = b \), this would be a version of induction. Here is another version.

\[
x := b ; x := 0
\]

\[
\text{while } i < x \text{ do
} x := f(i, x)
\text{ od}
\]

It would be interesting to see if we can treat

\[
\text{while } b(x) \text{ do
} x := f(x)
\text{ od}
\]

as a realizer in pure first-order logic. We will consider this further on Thursday.
We can have a more efficient form of induction if we used a "tail recursive" or iterative form. For example, we can define an iterative add that uses only a constant amount of space vs a linear amount.

\[
\text{add}(x, y) = \text{it-add}(x, y, y) \quad \text{where}
\]

\[
\text{it-add}(x, y, z) =
\]

\[
\begin{align*}
& \text{if } x = 0 \text{ then } 3 \\
& \text{else } \text{it-add}(x-1, y, z+1)
\end{align*}
\]

What is the iterative form of induction?