The λ-calculus can be reduced even further:

In the pure λ-calculus, there are no constants \( C \).

Thus, there is no \( \delta \)-reduction, only \( \beta \)-reduction.

The pure λ-calculus is still Turing-complete, i.e., one can implement any computable function in the pure λ-calculus.

Since there are no constants (and no \( \delta \)-rules), the constants have to be represented by suitable pure λ-terms whose \( \beta \)-reductions evaluates them in an appropriate way.

We have to choose a representation of data objects by λ-terms. Then one can implement all computable functions on these objects by λ-terms.

Data structure of natural numbers:

\[
\begin{align*}
\text{let } n \text{ the } n\text{-fold application of a function represents the number } n \\
\text{This representation should not depend on a particular } f \text{ or } x.
\end{align*}
\]

\( \Rightarrow \) We represent any natural number \( n \)
by the λ-term $\overline{n} = \lambda f. x. f^n x$

E.g.: $\overline{0} = \lambda f. x. x$

$\overline{1} = \lambda f. x. f x$

$\overline{2} = \lambda f. x. f (f x)$

... 

Now any computable fact. can be implemented as a λ-term
operating on this representation of nat. numbers. E.g.:

$\overline{\text{Succ}}$ should be a pure λ-term such that

for any $n \in \mathbb{N}$:

$\overline{\text{Succ}} \overline{n} \rightarrow^* \overline{n+1}$

How does $\overline{\text{Succ}}$ look like?

$\overline{\text{Succ}} \overline{n} = \overline{\text{Succ}} (\lambda f. x, f^n x) \rightarrow^\beta \lambda f. x. f (f^n x)$

To find $\overline{\text{Succ}}$, it is a good idea to transform
the desired result $\overline{n+1}$ such that it contains $\overline{n}$.

$\overline{n+1} = \lambda f. x. f^{n+1} x = \lambda f. x. f (f^n x) \equiv^\beta \lambda f. x. f (\overline{n} f x)$

$(\lambda f. x. f^n x) f x$
So if \( \text{Succ} \) is applied to \( \bar{n} \), the result should be \( \lambda f \, x. \, f(\bar{n} \, f \, x) \)

Solution \( \text{Succ} = \lambda n \, f \, x. \, f(\bar{n} \, f \, x) \)

\[
\begin{align*}
\text{Succ} \, \bar{n} &= (\lambda n \, f \, x. \, f(\bar{n} \, f \, x)) \, (\lambda f \, x. \, f^n \, x) \\
&\xrightarrow{\beta} \\
&\lambda f \, x. \, f((\lambda f \, x. \, f^n \, x) \, f \, x) \\
&\xrightarrow{\beta} \\
&\lambda f \, x. \, f(f^n \, x) \\
&\xrightarrow{\beta} \\
&\lambda f \, x. \, f^{n+1} \, x = \bar{n+1}
\end{align*}
\]

In a similar way, one can implement plus, times, \( \ldots \)

\( \Rightarrow \) any computable func. on \( \mathbb{N} \) can be implemented in this way.

Data structure of Booleans:

We choose the following representation of True and False by \( \lambda \)-terms:

\[
\begin{align*}
\text{True} &= \lambda x \, y. \, x \\
\text{False} &= \lambda x \, y. \, y
\end{align*}
\]

Now we can implement any computable func. on Booleans. \( \text{E.g.:} \)
\[ \begin{align*}
& \quad \quad \text{if True } x \quad \quad \longrightarrow^* \quad x \\
& \quad \quad \text{if False } x \quad \quad \longrightarrow^* \quad y \\
& \quad \quad \text{if should be} \quad \quad \lambda \ a \ y. \ a \ x \ y \\
& \quad \quad \text{or simpler} \quad \quad \lambda \ a. \ a
\end{align*} \]

Since pure \( \lambda \)-calculus is a complete prog. language, we can also implement "fix" as a pure \( \lambda \)-term:

\[ \text{fix should satisfy: fix } f \quad \longrightarrow^* \quad f \ (\text{fix } f) \]

This can be done by Turing's Fixpoint Combinator:

\[ \text{fix} : \quad (\lambda \ x \ y. \ y \ (x \ x \ y)) \ (\lambda \ x \ y. \ y \ (x \ x \ y)) \]

Drawbacks of pure \( \lambda \)-calculus:

- unreadable
- inefficient
- not suitable for type checking: \( \text{fix} \) can only be implemented by pure \( \lambda \)-terms that are not
well typed \Rightarrow any recursive Haskell-program would be compiled into a non-well-typed pure \lambda-term.