7. Considere a função in $= [\underline{0}, \mathsf{succ}]$ que exprime a forma como os números naturais são gerados a partir do número 0, de acordo com o diagrama seguinte,

$$\begin{tikzcd} 1 \arrow[r, "i_1"] \arrow[dr, "\underline{0}"'] & 1 + \mathbb{N}_0 \arrow[d, "\mathsf{in}=[\underline{0},\mathsf{succ}]"] & \mathbb{N}_0 \arrow[l, "i_2"'] \arrow[dl, "\mathsf{succ}"] \\ & \mathbb{N}_0 & \end{tikzcd} \tag{F9}$$

onde succ $n = n + 1$. Sabendo que o tipo 1 coincide com o tipo () em Haskell e é habitado por um único elemento, também designado por (), calcule a inversa de in,

$$\begin{array}{l} \mathsf{out}\ 0 = i_1\ () \\ \mathsf{out}\ (n+1) = i_2\ n \end{array}$$

resolvendo em ordem a out a equação out $\cdot$ in $= id$ e introduzindo variáveis. (NB: poderá deste cálculo inferir que in e out são isomorfismos? Justifique.)

Resolução

$out \cdot in = id$

{ def. in $= [\underline{0}, succ]$ }

$\equiv out \cdot [\underline{0}, succ] = id$

{ reflexão-+, **lei (19)** }

$\equiv out \cdot [\underline{0}, succ] = [i_1, i_2]$

{ fusão-+, **lei (20)** }

$\equiv [out \cdot \underline{0}, out \cdot succ] = [i_1, i_2]$

{ eq-+, **lei (27)** }

$$\equiv \begin{cases} out \cdot \underline{0} & = & i_1 \\ out \cdot succ & = & i_2 \end{cases}$$

{ igualdade extensional, **lei (71)** }

$$\equiv \begin{cases} (out \cdot \underline{0})\ n & = & i_1 n \\ (out \cdot succ)\ n & = & i_2\ n \end{cases}$$

{ **1** $= \{()\}$ }

$$\equiv \begin{cases} (out \cdot \underline{0})\ n & = & i_1 () \\ (out \cdot succ)\ n & = & i_2\ n \end{cases}$$

{ def-comp, **lei (72)** }

$$\equiv \begin{cases} out\ (\underline{0}\ n) & = & i_1 () \\ out\ (succ\ n) & = & i_2\ n \end{cases}$$

{ def-const, **lei (74)**; def. $succ = n + 1$ }

$$\equiv \begin{cases} out\ 0 & = & i_1() \\ out\ (n+1) & = & i_2\ n \end{cases}$$

Conclusão

in e out são inversas uma da outra e, como tal, isomorfismos. Ambas transformam elementos de $1 + \mathbb{N}_0$ em elementos de $\mathbb{N}_0$ e vice-versa sem perda de informação.

Haskell

In [1]:

```
out 0 = Left ()
out n = Right (n-1)

-- type checking
:t out
```

`out :: forall b. (Eq b, Num b) => b -> Either () b`

In [2]:

```
-- testing out with 3 and 0

out 3
out 0
```

```
Right 2
Left ()
```