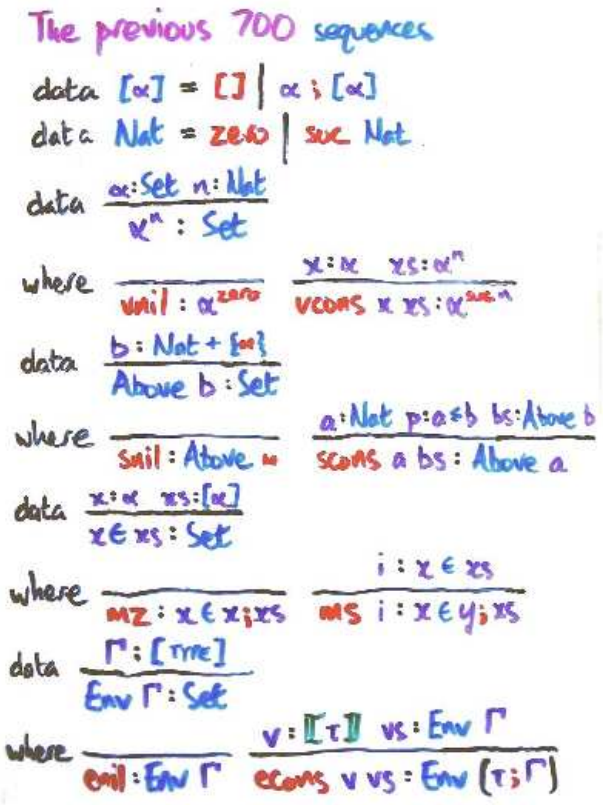
(Mc)Bride and Prejudice
(by a student of J. Austin)

Nimish Shah
University of York
nimish@cs.york.ac.uk

Dr McBride's Talk (York, November 2008)!



(http://sneezy.cs.nott.ac.uk/fun/nov-07/)

Haskell Version of $\mathbb{N}_0$

(Inspired by Runciman's "*What about the natural numbers?*" [4].)

```
-- Given an alphabet of 1 character (or generator).
data A = G0 deriving (Eq,Ord,Show)

n0 = []
n1 = G0 : n0
n2 = G0 : n1
n3 = G0 : n2
n4 = G0 : n3
```

In other words:

The number '4' is defined as `n4 = G0:G0:G0:G0:[]`

(Or `n4 = [G0,G0,G0,G0] :: [A]`)

Mathematical definition of $\mathbb{N}_0$

$$
\begin{aligned}
0 &=_{\text{def}} \emptyset &&=_{\text{def}} \{\} \\
1 &=_{\text{def}} 0 \cup \{0\} &&=_{\text{def}} \{0\} \\
2 &=_{\text{def}} 1 \cup \{1\} &&=_{\text{def}} \{0, 1\} \\
3 &=_{\text{def}} 2 \cup \{2\} &&=_{\text{def}} \{0, 1, 2\} \\
4 &=_{\text{def}} 3 \cup \{3\} &&=_{\text{def}} \{0, 1, 2, 3\}
\end{aligned}
$$

Here the set of natural numbers $\mathbb{N}_0 =_{\text{def}} \{0, 1, 2, 3, 4, \ldots\}$ is constructed from the empty set ($\emptyset$) and the *succ* operator, which in set theory [2, pg. 173] is defined as: $succ(x) =_{\text{def}} x \cup \{x\}$

Problem?

$$0 =_{\text{def}} \emptyset$$
$$1 =_{\text{def}} 0 \cup \{0\}$$
$$2 =_{\text{def}} 1 \cup \{1\}$$
$$3 =_{\text{def}} 2 \cup \{2\}$$
$$4 =_{\text{def}} 3 \cup \{3\}$$
$$\ldots$$
$$\{0, 1, 2, 3, 4, 5, \ldots\} :: \mathbb{N}_0$$

What is this saying?

```
type Nat = [A]
data A = G0 | G1 | G2 | G3

n0 = []
n1 = G0 : n0
n2 = G1 : n1
n3 = G2 : n2
n4 = G3 : n3
...
[G3, G2, G1, G0, G3, ...]::Nat
```

That this is not saying?

Answer

$\mathbb{N}_0 =_{\text{def}} \{0, 1, 2, 3, 4, 5, \ldots\}$

1. Countable infinite set.
2. Each element has **an** internal 'structure' (e.g., 3 is constructed from 19 symbols).
3. Pair-wise different for **all** pairs of elements.
4. Any pair can be well-ordered by $<$ (i.e., $\in$) or $\leq$ (i.e., $\subseteq$).

```
data A = G0 | G1 | G2 | G3
[G3,G2,G1,G0,G3,...] :: Nat
```

1. Countable infinite list.
2. Each element has **no** internal 'structure'.
3. Finite number of generators $\Rightarrow$ **not** pair-wise different for all possible pairs.
4. Does `[G3] == [G1]` (in length) $\Rightarrow$ `G3 == G1`?

Hypothesis

1. Colin & Conor use *cons* (`:`) when constructing their lists.
2. Why do functional programmers use cons?
   (Relic from Lisp (Finite lists & Pairs)? [3, Sec. 3])

Bold Step Into The Unknown

1. Think about functional programming like a mathematician.
2. Abandon cons and use list concatenation (`++`) as primitive.
3. Think of lists as semigroups/monoids.

Semigroups/Monoids (Grillet [1])

Assume:

1. Alphabet of finite number of generators/symbols/characters.
   `data A = G0 | G1 | G2 | G3` (say)
2. Words are finite lists of generators. `n3 = [G2, G1, G0]` (say)
3. A domain nat_0 is an infinite list of all finite words.
   i.e., `nat0` $=_{\texttt{def}}$ $[\texttt{n0}, \texttt{n1}, \texttt{n2}, \texttt{n3}, \ldots]$ (say)

1st Reward: (Solves Conor's problem)

1. Get `<nat0,++,[]>` as a monoid structure (Chapter 1 of Grillet).
2. Get a **natural pre-order** (Green's relations, chapter 2 of Grillet).

$$a \leq_{\mathrm{R}} b \text{ iff } \exists u \in \texttt{nat0}.\, au = b$$

(or)

$$a \leq_{\mathrm{L}} b \text{ iff } \exists v \in \texttt{nat0}.\, va = b$$

2nd Reward: (Solves Colin's problem)

1. Define $\sim$ to be the equivalence relation on the length of a list.
2. Let $nat_0/\sim$ be the quotient of nat_0 under $\sim$ i.e., partition
   `nat0 = [[], [G0], [G1], [G2], [G3], [G0,G0], ...]`
   into
   ```
   [ [ [] ],                              – a list of length 0 lists
     [ [G0], [G1], [G2], [G3] ],          – a list of length 1 lists
     [ [G0,G0], ..., [G3,G3] ],           – a list of length 2 lists
     ... ]
   ```
3. In nat_0, two lists are equal if and only if same length and same order of constructors (**pre-order** is dictionary/lexical ordering).
4. In $nat_0/\sim$, two lists are equal in length, i.e., belong to the same equivalence class. Hence `[G3]==[G1]` with `G3 /= G1`.

Summary of Talk

1. Abandon cons (`:`) and use concatenation (`++`) instead.
2. Treat **the data structure** of lists as semigroups/monoids.
3. At the type level divide lists into infinite and finite lists.

Question?

Does this mean that there is a way of thinking about **computation** (carried out in a functional style) as semigroups/monoids?

Answer

YES!!!! See forthcoming talks in the series: "*Monoids, Functional Programming, and Category Theory.*"

References

[1] P.A. Grillet. *Semigroups*. Marcel Dekker, 1995.

[2] S. Lipschutz. *Theory and problems of Set Theory and Related Topics*. Schaum's Outline Series. McGraw-Hill, 1964.

[3] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. Technical Report AIM-008, MIT, 1959.

[4] C. Runciman. What about the Natural Numbers? *Computer Languages*, 14(3):181–191, 1989.