Natural Numbers

We will use natural numbers to illustrate several ideas that will apply to *Haskell* data types in general.

For the moment we will ignore that fact that each type in *Haskell* includes $\perp$ as a possible value.

We note that it is only possible to define the natural numbers up to isomorphism.

1 Peano's Axioms

Let's consider a definition of *Natural Numbers*, $\mathbb{N}$, based on Peano's axioms.

1. $0 \in \mathbb{N}$
2. $m \in \mathbb{N} \Rightarrow succ(m) \in \mathbb{N}$.

Oops, we forgot to exclude negative numbers!

1. $0 \in \mathbb{N}$
2. $m \in \mathbb{N} \Rightarrow succ(m) \in \mathbb{N}$.
3. $m \in \mathbb{N} \Rightarrow succ(m) \neq 0$

We also need one more axiom.

1. $0 \in \mathbb{N}$
2. $m \in \mathbb{N} \Rightarrow succ(m) \in \mathbb{N}$.
3. $m \in \mathbb{N} \Rightarrow succ(m) \neq 0$
4. $succ(m) = succ(n) \Rightarrow m = n$

Oh my gosh! I just noticed that the nonnegative real numbers, the rational excluding negative integers, and lots of other sets, all satisfy these axioms. I forgot the induction axiom!

1. $0 \in \mathbb{N}$
2. $m \in \mathbb{N} \Rightarrow succ(m) \in \mathbb{N}$.
3. $m \in \mathbb{N} \Rightarrow succ(m) \neq 0$
4. $succ(m) = succ(n) \Rightarrow m = n$
5. For any predicate P in the first order predicate calculus,
   $(P(0) \wedge (P(m) \Rightarrow P(succ(m)))) \Rightarrow (\forall n \in \mathbb{N}.P(n))$

The induction axiom insures that $\mathbb{N}$ is the smallest set that satisfies the above axioms.

While you were looking at the previous page, I remembered that there are some completeness issues with the axioms for number theory. The axioms on the previous page are satisfied by an infinite number of sets, most of which are a bit weird.

We have to allow P to be any predicate.

1. $0 \in \mathbb{N}$
2. $m \in \mathbb{N} \Rightarrow succ(m) \in \mathbb{N}$.
3. $m \in \mathbb{N} \Rightarrow succ(m) \neq 0$
4. $succ(m) = succ(n) \Rightarrow m = n$
5. For any predicate P,
   $(P(0) \wedge (P(m) \Rightarrow P(succ(m)))) \Rightarrow (\forall n \in \mathbb{N}.P(n))$

Since we have no way of expressing more than a countable subset of the predicates on $\mathbb{N}$, perhaps it is better to restate the final axiom in terms of sets.

1. $0 \in \mathbb{N}$
2. $m \in \mathbb{N} \Rightarrow succ(m) \in \mathbb{N}$.
3. $m \in \mathbb{N} \Rightarrow succ(m) \neq 0$
4. $succ(m) = succ(n) \Rightarrow m = n$
5. $(X \subseteq \mathbb{N} \wedge 0 \in X \wedge (m \in X \Rightarrow succ(m) \in X)) \Rightarrow X = \mathbb{N}$

Are you really sure we finally have a correct definition for the natural numbers?

Are you even sure I finally managed to get my parentheses balanced correctly?

2 Natural Numbers defined using Category Theory

Let's try another approach.

Consider a category, US (for unary system), where

An Object is any triple (S, a, f) such that

- S is a set,
- $a \in s$, and
- $f : S \to S$ is a function.

An Arrow is any homomorphism between objects.

The desired unary system, $(\mathbb{N}, 0, succ)$, is an initial object in this category.

(An initial object is an object such that for any object, A, there is exactly one arrow from the initial object to A. In particular, there is exactly one arrow from the initial object to itself, which in this case is the identity function.)

We will define category, object, arrow and initial object later. Initial objects are unique up to isomorphism, when they exist, and in cases like this initial objects always exist.

For now, all we need to understand about this definition is that the natural numbers is a unary system such that for another unary system there is exactly one homomorphism from the natural numbers to the other unary system.

If (S, a, f) and (T, b, g) are any two unary systems, then a homomorphism, h, is a structure preserving function from S to T. That is

$$h : S \rightarrow T$$
$$h(a) = b$$
$$\forall s \in S.\, h(f(s)) = g(h(s))$$

Later we will give a more general definition of homomorphism that will apply to a variety of different structures.

To see that $(\mathbb{N}, 0, succ)$ is an initial object, it is sufficient to show that for any other object, (S, a, f), in US there is a unique homomorphism from $(\mathbb{N}, 0, succ)$ to (S, a, f).

Clearly, h defined by

$$h(0) = a$$
$$\forall n \in \mathbb{N}.h(succ(n)) = f(h(n))$$

is the unique homomorphism.

This is clearly the case because of our intuitive understanding of the natural numbers. We are actually using the requirement that the above holds to define the natural numbers.

Quiz: For each of the following unary systems, (i) find the unique homomorphism from $(\mathbb{N}, 0, succ)$ to the system, (ii) determine whether there are zero, one or more than one homomorphisms from the unary system to $(\mathbb{N}, 0, succ)$, (iii) find either two or all homomorphisms from the unary system to $(\mathbb{N}, 0, succ)$, and in the case of a unique homomorphism to $(\mathbb{N}, 0, succ)$ determine if the homomorphism is an isomorphism.

1. $(\mathbb{Z}, 0, succ)$
2. $(\mathbb{Z}^+, 1, succ)$
3. (Strings over $\{'a'\}$, $'a'$, $(+'a')$) where $+$ means concatenate.
4. $(\mathbb{Q}^+, 0, succ)$, where $succ\ x = x + 1$
5. $(\{True, False\}, False, \neg)$
6. $(\{0, 1, 2, 3, 4\}, 0, succ \pmod 5)$
7. $(\{0, 1, 2, 3, 4\}, 3, succ \pmod 5)$

Quiz:

1. Prove that if $h_1 : S_1 \rightarrow S_2$ is a homomorphism from (S_1, a_1, f_1) to (S_2, a_2, f_2) and $h_2 : S_2 \rightarrow S_3$ is a homomorphism from (S_2, a_2, f_2) to (S_3, a_3, f_3) then $h_2 \circ h_1$ is also a homomorphism.

2. Prove that if for each unary system there is a unique homomorphism from (S, a, f) to that unary system, then (S, a, f) is isomorphic to $(\mathbb{N}, 0, succ)$.

Note: (S, a, f) and $(\mathbb{N}, 0, succ)$ are isomorphic unary systems if there exist homomorphisms $g : S \rightarrow \mathbb{N}$ and $h : \mathbb{N} \rightarrow S$ such that $g \circ h = id_{\mathbb{N}}$ and $h \circ g = id_S$.

3 Haskell `Nats`

With *Bird's* definition of `Nat`s in Haskell (ignoring the `deriving` stuff),

```
data Nat = Zero | Succ Nat
```

we have an additional constructor, $\bot$.

Everything we have done so far carries over with minimal change. Instead of a unary system we have `(Nat, Zero, ⊥, Succ)`.

A homomorphism from $(\texttt{Nat}, \texttt{Zero}, \bot_{\texttt{Nat}}, \texttt{Succ})$ to $(S, a, \bot_S, f)$ is a function $h : \texttt{Nat} \to S$ such that

$$h\ \texttt{Zero} = a$$
$$h\ \bot_{\texttt{Nat}} = \bot_S$$
$$\forall n \in \texttt{Nat}. h\ (\texttt{Succ}\ n) = f(h\ n)$$

We will usually refer to $\perp$ without using a subscript, even though each domain (set of values corresponding to a type — more or less) has a different $\perp$. Sometimes we will skip explicitly mentioning $\perp$, since the impact of adding $\perp$ is straight forward.

The second condition on the previous page, $h\ \perp = \perp$, will usually be stated as "h is strict".

4 Catamorphisms

The homomorphisms from `Nat`, or more generally homomorphisms from types defined by *Haskell* `data` statements, are called *catamorphisms* .

The catamorphisms for a type are exactly the functions that can be defined using the $fold$ function for the type.

The $fold$ functions for `Nat`'s (given in *Bird*) is

```
foldn :: (a -> a) -> a -> Nat
foldn h c Zero = c
foldn h c (Succ n) = h (foldn h c n)
```

Recall that g is a homomorphism from $(\texttt{Nat}, \texttt{Zero}, \perp, \texttt{Succ})$ to $(S, a, \perp, f)$ iff

g is strict

$g\ \texttt{Zero} = a$

$\forall n \in \texttt{Nat}.g\ (\texttt{Succ}\ n) = f(g\ n)$

in which case

$g = \texttt{foldn}\ f\ a$

The third condition above, $\forall n \in \texttt{Nat}.g\ (\texttt{Succ}\ n) = f(g\ n)$, can be written more compactly as $g \circ \texttt{Succ} = f \circ g$.

Another way to look at

```
foldn :: (a -> a) -> a -> Nat
foldn h c Zero = c
foldn h c (Succ n) = h (foldn h c n)
```

is that `foldn h c n` constructs a new value by replacing the `Zero` constructor in `n` with `c` and each `Succ` constructor with `h`.

For example

```
foldn h c (Succ (Succ (Succ Zero))) = h (h (h c))
```

4.1 Fusion

Suppose

- `foldn g a` is a homomorphism from `(Nat,Zero,⊥,Succ)` to `(`S`,a,⊥,g)`,
- `foldn h b` is a homomorphism from `(Nat,Zero,⊥,Succ)` to `(`T`,b,⊥,h)`, and
- `f` is a homomorphism from `(`S`,a,⊥,g)` to `(`T`,b,⊥,h)`.

Since the composition of two homomorphisms is a homomorphism, and there is exactly one homomorphism from `(Nat,Zero,⊥,Succ)` to `(`T`,b,⊥,h)`, it must be the case that:

```
f . foldn g a = foldn h b
```

The result on the previous page is the *fusion Law for* `Nat` .

In the book, the condition that `f` is a homomorphism is stated as

`f` is strict

```
f a = b
f.g = h.f
```

It may help to note that

```
Zero: Nat
Succ: Nat -> Nat

a: S
g: S -> S

b: T
h: T -> T

f: S -> T
```

4.2 Induction

In *Bird*, the fusion law for `Nat`s was proved using induction.

We will reverse this process and show, using the fusion law for `Nat`s, that proof by induction is valid on $\mathbb{N}$.

Before going into detail, let us consider why we can prove the validity of proof by induction, while mathematicians regard induction as an axiom.

Recall that on page 11 we ***defined*** the natural numbers to be the initial object of the category of unary systems. It turns out that this is equivalent to the induction axiom.

We will break our proof into a number of small lemmas so that each lemma and its proof can be written on a single page.

We will use the following definitions throughout the lemmas and the theorem that follows.

```
f1 n      = (n, True)
f2 n      = (n, p n)
g         = Succ
h (n, q)  = (Succ n, q || p (Succ n))
a         = Zero
b1        = f1 a = (Zero, True)
b2        = f2 a = (Zero, p Zero)
```

While our work is done using the notation of *Haskell*, we note that the results will still hold even if `p` is not a computable predicate.

Important: In our consideration of natural numbers, we will restrict our attention to $\mathbb{N}$. In particular, we will not allow numbers to be $\perp$ or partial numbers, or infinite. This seems reasonable, since induction is usually expressed in terms of $\mathbb{N}$ rather than `Nat`.

Furthermore, we will assume that `p` is a total predicate (i.e. `(p n)` can never be $\perp$). This seems reasonable, since we are not restricting `p` to be a computable predicate.

Lemma 1 `foldn Succ Zero = id`

Proof: The function `foldn Succ Zero` is a homomorphism from $\mathbb{N}$ to $\mathbb{N}$. The identity function is also a homomorphism from $\mathbb{N}$ to $\mathbb{N}$. Since there can be only one such homomorphism, the result holds. □

Notice that we avoided using induction, since that would result in a circular argument.

Lemma 2 *If* `p n` $\Rightarrow$ `p (Succ n)`

then `p (Succ n) = (p n || p (Succ n))`

Proof: If `p n = True` then

```
p (Succ n)
```

$\equiv$ (assumptions: `p n = True` and `p n` $\Rightarrow$ `p (Succ n)`)

```
True
```

$\equiv$ (algebra of logic)

```
True || p (Succ n)
```

$\equiv$ (asumption: `p n = True`)

```
p n || p (Succ n)
```

Otherwise (`p n = False`), by a similar argument

```
p (Succ n) = False || p (Succ n)
 = p n || p (Succ n)
```

□

If we allow `p n` to be $\perp$ then *Lemma 2* will no longer hold.

However, we can define a predicate q, which is not computable, by

$$q\, n \equiv \begin{cases} \texttt{True}, & \text{if } \texttt{p n} \equiv \texttt{True} \\ \texttt{False}, & \text{if } \texttt{p n} \equiv \texttt{False} \\ \texttt{False}, & \text{if } \texttt{p n} \equiv \perp \end{cases}$$

Now, if $\forall n \in \mathbb{N}.q\, n$ then $\forall n \in \mathbb{N}.p\, n$.

Lemma 3 *f1.g = h.f1*

Proof: For any $n \in \mathbb{N}$

```
(f1.g) n
   ≡ (plug and grind)
(Succ n, True)
   ≡ (algebra of logic)
(Succ n, True || p (Succ n))
   ≡ (plug and grind)
(h.f1) n
```

so the result holds by extensionality. □

Lemma 4 *f1 = foldn h (Zero, True)*

Proof:

```
f1
    ≡ (Lemma 1)
f1 . foldn Succ Zero
    ≡ (fusion; Lemma 3)
foldn h (f1 Zero)
    ≡ (plug and grind)
foldn h (Zero, True)
```

The fusion step uses:

```
f1.foldn Succ Zero = foldn h (f1 Zero)
```

□

Lemma 5 *If* `p n ⇒ p (Succ n)` *then* `f2.g = h.f2`

Proof: For any `n∈ ℕ`

```
(f2.g) n
   ≡ (plug and grind)
(Succ n, p (Succ n))
   ≡ (Lemma 2)
(Succ n, p n || p (Succ n))
   ≡ (plug and grind)
(h.f2) n
```

so the result holds by extensionality. □

Lemma 6 *If* $(p\ Zero) \wedge (p\ n \Rightarrow p\ (Succ\ n))$

then `f2 = foldn h (Zero, True)`

Proof:

```
f2
   ≡ (Lemma 1)
f2 . foldn Succ Zero
   ≡ (fusion; Lemma 5)
foldn h (f2 Zero)
   ≡ (p Zero is True)
foldn h (Zero, True)
```

The fusion step uses:

```
f2.foldn Succ Zero = foldn h (f2 Zero)
```

□

Theorem 1 (Induction) *If* $(p\ Zero) \wedge (p\ n \Rightarrow p\ (Succ\ n))$ *then* $\forall n \in \mathbb{N}.p\ n = True$

Proof: On $\mathbb{N}$ (but not `Nat`), by *Lemma 4* and *Lemma 6*,

```
f1 = f2
```

so

```
snd.f1 = snd.f2
```

Since

```
(snd.f1) n = True
(snd.f2) n = p n
```

We have the desired result, that for $\forall n \in \mathbb{N}.p\ n = True$.

□

We will now demonstrate the use of fusion to do the classic undergraduate problem of showing

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$$

We will use `Integer`s to represent $\mathbb{N}$s, with the following fold function.

```
foldi :: (a -> a) -> a -> Integer -> a
foldi g a 0 = a
foldi g a (n+1) = g (foldi g a n)
```

That is, we will use the concrete type `Integer` to represent the abstract type $\mathbb{N}$. Negative `Integer`s will not be used, and `foldi` is the concrete implemetation of an abstract `fold`$\mathbb{N}$ function. If we used `Nat`s to represent $\mathbb{N}$s, then we would need to ignore partial and infinite `Nat`s, and use `foldn` to implement `fold`$\mathbb{N}$.

To recursively define

$$h\, n = (n,\ \textstyle\sum_{i=0}^{n} i)$$

we can define and use `g`

```
g :: (Integer,Integer) -> (Integer,Integer)
g (a, b) = (a+1, b + (a+1))
```

so

```
h 0     = (0, 0)
```
$= (0,\ \sum_{i=0}^{0} i)$

```
h (n+1) = g (h n)
```
$= \mathtt{g}\,(n,\ \sum_{i=0}^{n} i) = (n+1,\ \sum_{i=0}^{n+1} i)$

or

```
h n = foldi g (0, 0) n
```
$= (n, \sum_{i=0}^{n} i)$

recalling that

```
foldi :: (a -> a) -> a -> Integer -> a
foldi g a 0 = a
foldi g a (n+1) = g (foldi g a n)
```

We now define

```
f :: Integer -> (Integer,Integer)
f n = (n, n*(n+1) `div` 2)
```

and use fusion to show that `f = foldi g (0, 0)` from which the desired result follows immediately.

Since

```
n*(n+1) `div` 2 + (n+1) = (n+1)*(n+2) `div` 2
```

we see that

```
(g.f) n
   ≡ (plug and grind)
g (n, n*(n+1) `div` 2)
   ≡ (plug and grind)
(n+1, (n+1)*(n+2) `div` 2)
   ≡ (plug and grind)
(f.(+1)) n
```

or

```
g.f = f.(+1)
```

Now we have

```
f
   ≡ (foldi (+1) 0 = id)
f . foldi (+1) 0
   ≡ (fusion)
foldi g (0, 0)
```

In the fusion step we use

1. `f 0 = (0, 0)`
2. `g.f = f.(+1)`

However, since we are working with $\mathbb{N}$s rather than `Nat`s, we do not have the requirement that `f` be strict.

What we have show is that both `f` and `foldi g (0,0)` are the unique homomorphism from `(`$\mathbb{N}$`,0,(+1))` to `(`$\mathbb{N} \times \mathbb{N}$`, (0,0), g)` and therefore are equal.

Since

```
f n = (n, n*(n+1) `div` 2)
```

$$\texttt{foldi g (0, 0) n} = (n, \sum_{i=0}^{n} i)$$

we have

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$$

In an induction proof we would have

Basic Step: `f 0 = (0, 0)`

Induction Step: `h (n+1) = g (h n) = g (f n) = f (n+1)`

Notice how closely these correspond to our requirements for fusion to hold.

In the fusion step we use

1. `f 0 = (0, 0)`
2. `g.f = f.(+1)`