FIXED POINTS AND DIAGONAL ARGUMENTS

BARTOSZ MILEWSKI

What does Gödel's incompletness theorem, Russel's paradox, Turing's halting problem, and Cantor's diagonal argument have to do with the fact that negation has no fixed point? The surprising answer is that they are all related through Lawvere's fixed point theorem.

Before we dive deep into category theory, let's unwrap this statement from the point of view of a (Haskell) programmer. Let's start with some basics. Negation is a function that flips between `True` and `False`:

```
not :: Bool -> Bool
not True  = False
not False = True
```

A fixed point is a value that doesn't change under the action of a function. Obviously, negation has no fixed point. There are other functions with the same signature that have fixed points. For instance, the constant function:

```
true True  = True
true False = True
```

has `True` as a fixed point.

All the theorems I listed in the preamble (and a few more) are based on a simple but extremely powerful proof technique invented by Georg Cantor called the diagonal argument. I'll illustrate this technique first by showing that the set of binary sequences is not countable.

1. Cantor's job interview question

A binary sequence is an infinite stream of zeros and ones, which we can represent as Booleans. Here's the definition a sequence (it's just like a list, but without the nil constructor), together with two helper functions:

```
data Seq a = Cons a (Seq a)
  deriving Functor

head :: Seq a -> a
head (Cons a as) = a

tail :: Seq a -> Seq a
tail (Cons a as) = as
```

And here's the definition of a binary sequence:

```
type BinSeq = Seq Bool
```

If such sequences were countable, it would mean that you could organize them all into one big (infinite) list. In other words we could implement a sequence generator that generates every possible binary sequence:

```
allBinSeq :: Seq BinSeq
```

Suppose that you gave this problem as a job interview question, and the job candidate came up with an implementation. How would you test it? I'm assuming that you have at your disposal a procedure that can (presumably in infinite time) search and compare infinite sequences. You could throw at it some obvious sequences, like all `True`, all `False`, alternating `True` and `False`, and a few others that came to your mind.

What Cantor did is to use the candidate's own contraption to produce a counter-example. First, he extracted the diagonal from the sequence of sequences. This is the code he wrote:

```
diag :: Seq (Seq a) -> Seq a
diag seqs = Cons (head (head seqs)) (diag (trim seqs))

trim :: Seq (Seq a) -> Seq (Seq a)
trim seqs = fmap tail (tail seqs)
```

You can visualize the sequence of sequences as a two-dimensional table that extends to infinity towards the right and towards the bottom.

```
T F F T ...
T F F T ...
F F T T ...
F F F T ...
...
```

Its diagonal is the sequence that starts with the fist element of the first sequence, followed by the second element of the second sequence, third element of the third sequence, and so on. In our case, it would be a sequence `T F T T ...`.

It's possible that the sequence, `diag allBinSeq` has already been listed in `allBinSeq`. But Cantor came up with a devilish trick: he negated the whole diagonal sequence:

```
tricky = fmap not (diag allBinSeq)
```

and ran his test on the candidate's solution. In our case, we would get `F T F F ...` The tricky sequence was obviously not equal to the first sequence because it differed from it in the first position. It was not the second, because it differed (at least) in the second position. Not the third either, because it was different in the third position. You get the gist...

2. Power sets are big

In reality, Cantor did not screen job applicants and he didn't even program in Haskell. He used his argument to prove that real numbers cannot be enumerated.

But first let's see how one could use Cantor's diagonal argument to show that the set of subsets of natural numbers is not enumerable. Any subset of natural numbers can be represented by a sequence of Booleans, where `True` means a given number is in the subset, and `False` that it isn't. Alternatively, you can think of such a sequence as a function:

```
type Chi = Nat -> Bool
```

called a *characteristic* function. In fact characteristic functions can be used to define subsets of any set:

```
type Chi a = a -> Bool
```

In particular, we could have defined binary sequences as characteristic functions on naturals:

```
type BinSeq = Chi Nat
```

As programmers we often use this kind of equivalence between functions and data, especially in lazy languages.

The set of all subsets of a given set is called a *power set*. We have already shown that the power set of natural numbers is not enumerable, that is, there is no function:

```
enumerate :: Nat -> Chi Nat
```

that would cover all characteristic functions. A function that covers its codomain is called surjective. So there is no surjection from natural numbers to all sequences of natural numbers.

In fact Cantor was able to prove a more general theorem: for any set, the power set is always larger than the original set. Let's reformulate this. There is no surjection from the set A to the set of functions $A \to 2$ (where 2 stands for the two-element set of Booleans).

To prove this, let's be contrarian and assume that there is a surjection:

```
enumP :: A -> Chi A
```

Since we are going to use the diagonal argument, it makes sense to uncurry this function, so it looks more like a table:

```
g :: (A, A) -> Bool
g = uncurry enumP
```

Diagonal entries in the table are indexed using the following function:

```
delta :: a -> (a, a)
delta a = (a, a)
```

We can now define our custom characteristic function by picking diagonal elements and negating them, as we did in the case of natural numbers:

```
tricky :: Chi A
tricky = not . g . delta
```

If `enumP` is indeed a surjection, then it must produce our function `tricky` for some value of `x :: A`. In other words, there exists an `x` such that `tricky` is equal to `enumP x`. This is an equality of functions, so let's evaluate both functions at `x` (which will evaluate `g` at the diagonal).

```
tricky x == (enumP x) x
```

The left hand side is equal to:

```
tricky x = {- definition of tricky -}
not (g (delta x)) = {- definition of g -}
not (uncurry enumP (delta x)) = {- uncurry and delta -}
not ((enumP x) x)
```

So our assumption that there exists a surjection `A -> Chi A` led to a contradition!

```
not ((enumP x) x) == (enumP x) x
```

3. Real numbers are uncountable

You can kind of see how the diagonal argument could be used to show that real numbers are uncountable. Let's just concentrate on reals that are greater than zero but less than one. Those numbers can be represented as sequences of decimal digits (the digits following the decimal point). If these reals were countable, we could list them one after another,

just like we attempted to list all streams of Booleans. We would get some kind of a table infinitely extending to the right and towards the bottom. There is one small complication though. Some numbers have two equivalent decimal representations. For instance 0.48 is the same as 0.47999..., with infinitely many nines. So lets remove all rows from our table that end in an infinity of nines. We get something like this:

```
3 5 0 5 ...
9 9 0 8 ...
4 0 2 3 ...
0 0 9 9 ...
...
```

We can now apply our diagonal argument by picking one digit from every row. These digits form our diagonal number. In our example, it would be `3 9 2 9`.

In the previous proof, we negated each element of the sequence to get a new sequence. Here we have to come up with some mapping that changes each digit to something else. For instance, we could add one to it, modulo nine. Except that, again, we don't want to produce nines, because we could end up with a number that ends in an infinity of nines. But something like this will work just fine:

```
h n = if n == 8
      then 3
      else (n + 1) `mod` 9
```

The important part is that our function `h` replaces every digit with a different digit. In other words, `h` *doesn't have a fixed point.*

4. Lawvere's fixed point theorem

And this is what Lawvere realized: The diagonal argument establishes the relationship between the existence of a surjection on the one hand, and the existence of a no-fix-point mapping on the other hand. So far it's been easy to find a no-fix-point functions. But let's reverse the argument: If there is a surjection $A \to (A \to Y)$ then every function $Y \to Y$ must have a fixed point. That's because, if we could find a no-fixed-point function, we could use the diagonal argument to show that there is no surjection.

But wait a moment. Except for the trivial case of a function on a one-element set, it's always possible to find a function that has no fixed point. Just return something else than the argument you're called with. This is definitely true in *Set*, but when you go to other categories, you can't just construct morphisms willy-nilly. Even in categories where morphisms are functions, you might have constraints, such as continuity or smoothness. For instance, every continuous function from a real segment to itself has a fixed point (Brouwer's theorem).

As usual, translating from the language of sets and functions to the language of category theory takes some work. The tricky part is to generalize the definition of a fixed point and surjection.

4.1. Points and string diagrams.

First, to define a fixed point, we need to be able to define a point. This is normally done by defining a morphism from the terminal object 1, such a morphism is called a *global element*. In *Set*, the terminal object is a singleton set, and a morphism from a singleton set just picks an element of a set.

Since things will soon get complicated, I'd like to introduce string diagrams to better visualise things in a cartesian category. In string diagrams lines correspond to objects and

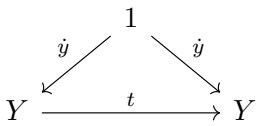
dots correspond to morphisms. You read such diagrams bottom up. So a morphism

$$\dot{a}\colon 1 \to A$$

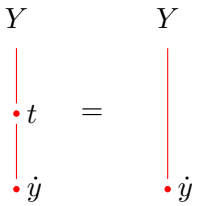
can be drawn as:

I will use dotted letters to denote "points" or morphisms originating in the unit. It is also customary to omit the unit from the picture. It turns out that everything works just fine with implicit units.

A fixed point of a morphism $t\colon Y \to Y$ is a global element $\dot{y}\colon 1 \to Y$ such that $t \circ \dot{y} = \dot{y}$. Here's the traditional diagam:

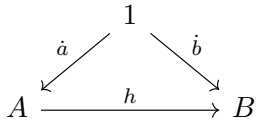


And here's the corresponding string diagram that encodes the commuting condition.

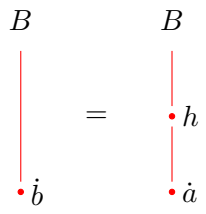


In string diagrams, morphisms are composed by stringing them along lines in the bottom up direction.

Surjections can be generalized in many ways. The one that works here is called *surjection on points*. A morphism $h\colon A \to B$ is surjective on points when for every point $\dot{b}$ of B (that is a global element $\dot{b}\colon 1 \to B$) there is a point $\dot{a}$ of A (the domain of h) that is mapped to $\dot{b}$. In other words $h \circ \dot{a} = \dot{b}$



Or string-diagrammatically, for every $\dot{b}$ there exists an $\dot{a}$ such that:



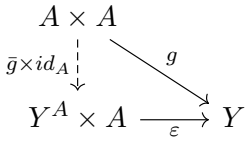
4.2. **Currying and uncurrying.** To formulate Lawvere's theorem, we'll replace B with the exponential object Y^A, that is an object that represents the set of morphisms from A to Y. Conceptually, those morphism will correspond to rows in our table (or characteristic functions, when Y is 2). The mapping:

$$\bar{g}\colon A \to Y^A$$

generates these rows. I will use barred symbols, like $\bar{g}$ for curried morphisms, that is morphisms to exponentials. The object A serves as the source of indexes for enumerating the rows of the table (just like the natural numbers in the previous example). The same object also provides indexing within each row.

This is best seen after uncurrying $\bar{g}$ (we assume that we are in a cartesian closed category). The uncurried morphism, $g\colon A \times A \to Y$ uses a product $A \times A$ to index simultaneously into rows and columns of the table, just like pairs of natural numbers we used in the previous example.

The currying relationship between these two is given by the universal construction:

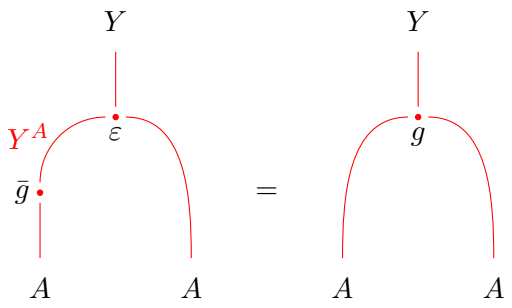


with the following commuting condition:

$$g = \varepsilon \circ (\bar{g} \times id_A)$$

Here, ε is the evaluation natural transformation (the couinit of the currying adjunction, or the dollar sign operator in Haskell).

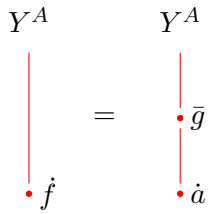
This commuting condition can be visualized as a string diagram. Notice that products of objects correspond to parallel lines going up. Morphisms that operate on products, like ε or g, are drawn as dots that merge such lines.



We are interested in mappings that are point-surjective. In this case, we would like to demand that for every point $\dot{f}\colon 1 \to Y^A$ there is a point $\dot{a}\colon 1 \to A$ such that:

$$\dot{f} = \bar{g} \circ \dot{a}$$

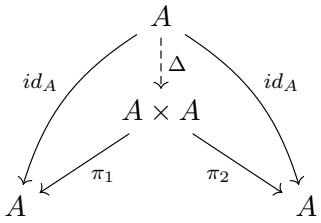
or, diagrammatically, for every $\dot{f}$ there exists an $\dot{a}$ such that:



Conceptually, $\dot{f}$ is a point in Y^A, which represents some arbitrary function $A \to Y$. Surjectivity of $\bar{g}$ means that we can always find this function in our table by indexing into it with some $\dot{a}$.

This is a very general way of expressing what, in Haskell, would amount to: Every function `f :: A -> Y` can be obtained by partially applying our `g_bar :: X -> A -> Y` to some `x :: X`.

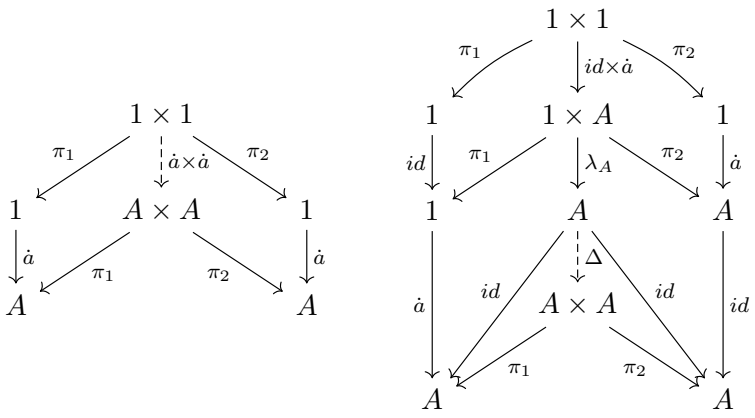
4.3. **The diagonal.** The way to index the diagonal of our table is to use the diagonal morphism $\Delta\colon A \to A \times A$. In a cartesian category, such a morphism always exists. It can be defined using the universal property of the product:



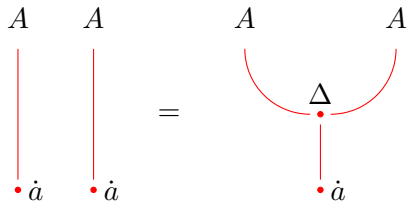
By combining this diagram with the diagram that defines the lifting of a pair of points $\dot{a}$ we arrive at a useful identity:

$$\dot{a} \times \dot{a} = \Delta_A \circ \lambda_A \circ (id_A \times \dot{a})$$

where λ_A is the left unitor, which asserts the isomorphism $1 \times A \to A$



Here's the string diagram representation of this identity:



In string diagrams we ignore unitors (as well as associators). Now you see why I like string diagrams. They make things much simpler.

4.4. **Lawvere's fixed point theorem.**

Theorem 4.1 (Lawvere)**.** *In a cartesian closed category, if there exists a point-surjective morphism $\bar{g}\colon A \to Y^A$ then every endomorphism of Y must have a fixed point.*

Note: Lawvere actually used a weaker definition of point surjectivity.

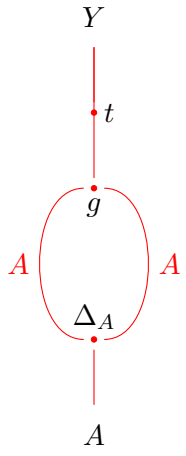
The proof is just a generalization of the diagonal argument.

Suppose that there is an endomorphims $t\colon Y \to Y$ that has no fixed point. This generalizes the negation of the original example. We'll create a special morphism by combining the diagonal entries of our table, and then "negating" them with t.

The table is described by (uncurried) g; and we access the diagonal through Δ_A. So the tricky morphism $A \to Y$ is just:

$$f = t \circ g \circ \Delta_A$$

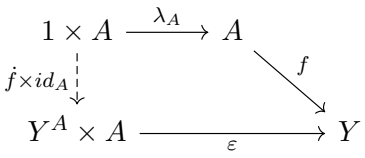
or, in diagramatic form:



$$f\colon A \xrightarrow{\Delta_A} A \times A \xrightarrow{g} Y \xrightarrow{t} Y$$

Since we were guaranteed that our table g is an exhaustive listing of all morphisms $A \to Y$, our new morphism f must be somewhere there. But in order to search the table, we have to first convert f to a point in the exponential object Y^A.

There is a one-to-one correspondence between points $\dot{f}\colon 1 \to Y^A$ and morphisms $f\colon A \to Y$ given by the universal property of the exponential (noting that $1 \times A$ is isomorphic to A through the left unitor, $\lambda_A\colon 1 \times A \to A$).



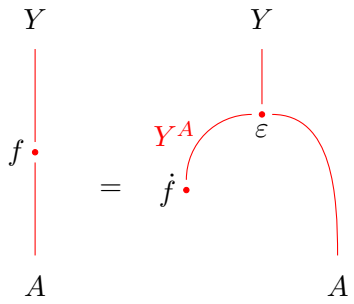
In other words, $\dot{f}$ is the curried form of $f \circ \lambda_A$, and we have the following commuting codition:

$$f \circ \lambda_A = \varepsilon \circ (\dot{f} \times id_A)$$

Since λ is an isomorphism, we can invert it, and get the following formula for f in terms of $\dot{f}$:

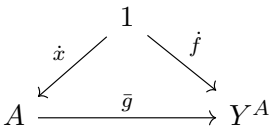
$$f = \varepsilon \circ (\dot{f} \times id_A) \circ \lambda_A^{-1}$$

In the corresponding string diagram we omit the unitor altogether.



Now we can use our assumption that $\bar{g}$ is point surjective to deduce that there must be a point $\dot{x} \colon 1 \to A$ that will produce $\dot{f}$, in other words:

$$\dot{f} = \bar{g} \circ \dot{x}$$



So $\dot{x}$ picks the row in which we find our tricky morphism. What we are interested in is "evaluating" this morphism at $\dot{x}$. That will be our paradoxical diagonal entry. By construction, it should be equal to the corresponding point of f, because this row is point-by-point equal to f; after all, we have just found it by searching for f! On the other hand, it should be different, because we've build f by "negating" diagonal entries of our table using t. Something has to give and, since we insist on surjectivity, we conclude that t is not doing its job of "negating." It must have a fixed point at $\dot{x}$.

Let's work out the details.

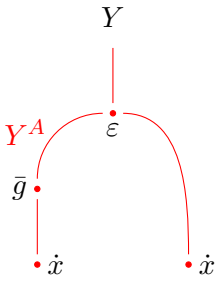
First, let's apply the function we've found in the table at row $\dot{x}$ to $\dot{x}$ itself. Except that what we've found is a point in Y^A. Fortunately we have figured out earlier how to get f from $\dot{f}$. We apply the result to $\dot{x}$:

$$f \circ \dot{x} = \varepsilon \circ (\dot{f} \times id_A) \circ \lambda_A^{-1} \circ \dot{x}$$

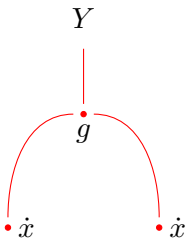
Plugging into it the entry $\dot{f}$ that we have found in the table, we get:

$$f \circ \dot{x} = \varepsilon \circ ((\bar{g} \circ \dot{x}) \times id_A) \circ \lambda_A^{-1} \circ \dot{x}$$

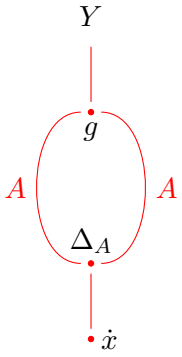
Here's the corresponding string diagram:



We can now uncurry $\bar{g}$

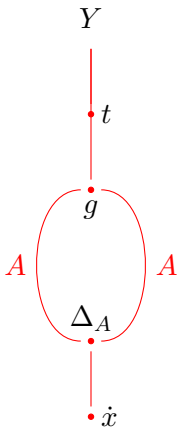


And replace a pair of $\dot{x}$ with a Δ:



Compare this with the defining equation for f, as applied to $\dot{x}$:

$$f \circ \dot{x} = t \circ g \circ \Delta_A \circ \dot{x}$$



In other words, the morphism $1 \to Y$:

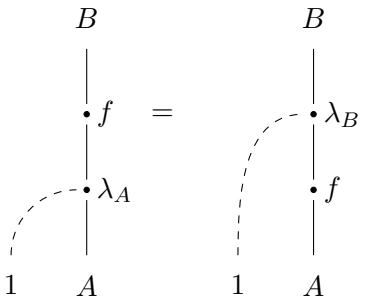
$$g \circ \Delta_A \circ \dot{x}$$

is a fixed point of t. This contradicts our assumption that t had no fixed point.

5. Conclusion

When I started writing this blog post I though it would be a quick and easy job. After all, the proof of Lawvere's theorem takes just a few lines both in the original paper and in all other sources I looked at. But then the details started piling up. The hardest part was convincing myself that it's okay to disregard all the unitors. It's not an easy thing for a programmer, because without unitors the formulas don't "type check." The left hand side may be a morphism from $A \times I$ and the right hand side may start at A. A compiler would reject such code. I tried to do due diligence as much as I could, but at some point I decided to just go with string diagrams. In the process I got into some interesting discussions and even posted a question on Math Overflow. Hopefully it will be answered by the time you read this post.

One of the things I wasn't sure about was if it was okay to slide unitors around, as in this diagram:



It turns out this is just naturality condition for λ, as John Baez was kind to remind me on Twitter (great place to do category theory!).

6. Acknowledgments

I'm grateful to Derek Elkins for reading the draft of this post.

7. Literature

- F. William Lawvere, Diagonal arguments and cartesian closed categories
- Noson S. Yanofsky, A Universal Approach to Self-Referential Paradoxes, Incompleteness and Fixed Points
- Qiaochu Yuan, Cartesian closed categories and the Lawvere fixed point theorem