MARKOV FUNCTIONS

We can use Markov algorithms — or something very close to them — to operate on numerals and thereby define strange functions from natural numbers to natural numbers.

For instance, we define $f := 29 \to 3; 2 \to 7; 72 \to 9$. These are substring replacement rules, sometimes called "productions." So f is an ordered list or finite sequence of productions. To compute f, we apply only the first applicable production, if there is one. We also always apply that production as far to the left as possible. If no production applies, f just returns its argument.

So $f(129) = 13$. Note that we did **not** apply the production $2 \to 7$. This is because we had an opportunity to apply $29 \to 3$, which comes first in the list.

Also $f(202) = 702$, because we have to replace the first 2 rather than the second.

Let $f^n(x)$ be the composition of f with itself n times. So $f^2(322) = f(f(322)) = f(372) = 39$.

What we are *most* interested in is a "runaway" process, where we keep applying f until no production is possible. This gives us, more or less, a Markov algorithm on numerals, which may not terminate.

We let $t_0 = x$. Then $t_{n+1} = f(t_n)$. The algorithm terminates if and when $t_{n+1} = f(t_n)$. In other words, the algorithm continues until a fixed point for f is discovered among the t_i. If this happens, then we can define $f^*(x) = t_n$. So f^* is the partial recursive function defined in terms of the total recursive "step" function f.

For instance, let's define $f = 2 \to 3; 3 \to 4;$. Then the attempted calculation of $f^*(2)$ gives us the finite tape sequence $2 \to 3 \to 4 \to 4$. We stop whenever we see the same n immediately next to itself. So $f^*(2) = 4$. In the usual language, 2 is in the domain of f^*.

Let's see at least non-terminating calculation. Let $f = 2 \to 3; 3 \to 2;$. Then the attempted calculation of $f^*(3)$ gives us $3 \to 2 \to 3 \to 2 \to ...$ This is an obvious infinite loop. We can use $f^*(3) = \infty$ to indicate a case like this where we are sure that the calculation does not terminate. From my constructivist viewpoint, we have 3 situations. We can believe that a calculation has or will terminate. We can believe that a calculation will not terminate. And the situation can be indeterminate. My objection to speaking of "returns or not" is that we don't have and don't ever expect

to have a total function that decides the situation one way or another. In particular cases, we can and do reason our way to a strong belief. But we don't have a general method. So RETURNS is not a full-fledged decidable/objective predicate.

MOTIVATION

A **tape machine** is approximately a Markov algorithm in the style of Turing machines. We used an alphabet of letters for "states" and stylized ones and zeros for "data."

For instance, let the program step function be $f = f_0| \to OOf_0; f_0 \to |$. The convention was to start with a particular state symbol f_0 at the beginning of the tape. It's actually easiest to just stick with f_i for names of "states" for when it comes time to encode these. The tape is just an always-finite string, without "infinite blanks" stretching out in one or both directions. The productions allow for what amounts to insertion and deletion.

Let's try to compute $f^*(|||)$.

$$f_0||| \to OOf_0|| \to OOOO| \to OOOOOOf_0 \to OOOOOO| \to OOOOOO|.$$

So $f^*(|||) = OOOOOO|$, because we've found a fixed point. There's no rule against using productions that operate on O and $|$, but the "Turing machine style" basically comes from using "letter symbols" to "do the work."

I wrote a C program to implement tape machines on the level of individual bits of computer memory. To do this, I encoded 0 as 01, 1 as 011, f_0 as 0111 and so on. I reserved the symbol 0 for punctuation, so that I could write a universal tape machine. I needed a way to indicate the end of the program to simulate and the data to feed it as input.

I wrote tape machine programs for calculating the nth Fibonacci number and for implementing cellular automata, etc. The largest tape machine program that I wrote (and the most recent) was a universal tape machine. This had around 200 productions.

I can't recall the details, but I've read about expressing Turing machine programs as recursive functions that operate on single integers. Basically the entire tape is encoded as a number, and the numerical code for the updated tape is returned.

I asked myself how I might do that with a tape machine program.

Keep in mind that my encoding on the level of bits already got me from

$f_0|O$ to 011101101. My encoding meant that I would be stuck with leading zeros, so I thought of reversing the tape before treating it as a number. This works if we ignore the punctuation symbol or just use 0111 for punctuation and bump f_0 up to 01111, etc.

So I started to think about how to use multiplication and addition to encode the production rules. I believe that it is possible, but to me it looks difficult and tedious. So I was motivated like the fox too lazy to climb for the grapes. What is so magical about additional and multiplication ? Practical, yes. But are they the essence of algorithm ? Indeed, how do children learn to add and multiply ? They operate on symbols. Which is of course the point of Markov's algorithm in the first place.[1]

Instead of worrying about reversing strings and translating productions into addition and multiplication, I could use encode in a way that avoided leading zeros. In my C program, I also encoded the productions of the step function in the same style. The primary point is that symbols like f_0 and f_1 were just human-convenient stand-ins for 0111 and 01111, and so on. So everything was just operating on binary numbers. I just needed to avoid leading zero, and my step function was *already* a numerical recursive function.

This is a simple idea, but the analogue of this for Turing machines was messier, because of the "infinite tape."

We end up with a situation where the step function program is just a binary number. Each instantaneous description of the tape is just a binary number. So instead of "encoding" the tape as a number, it's already a number. The productions, when written in machine code, are numbers that operate on numbers in a way that's generally nothing like addition and multiplication but just as intuitively algorithmic. Basically we "generalize" addition and multiplication to all feasible manipulations of numerals.

[1] As an anti-platonist in my philosophy of math, I was also ideologically motivated. I see numbers as something like equivalence classes of "generalized" numerals.Someone might object that a Markov function (as defined in the first part of this paper) is not really a function from N to N, because it will give different results depending on whether n is given in binary or decimal, etc. This is a valid concern, but we can simply specify that it operates on n expressed in particular base.