DMP Notes K. Sutner

Conway's Fractran

2023/05/26 v0.3

1 Programming with Fractions

Here is a slightly goofy model of computation [1] that shows that universal computation can be achieved by grade school arithmetic and a tiny bit of control-flow logic. Universal in this context means: every possible computation whatsoever can be carried out in this framework, in the sense that we can simulate an arbitray program, written in any language you like. The internal workings will be different, but our simulator has the exact same input/output behavior. Efficiency is of no interest here whatsoever.

The serious point is that systems that appear extremely weak still may support universal computation: it lurks in places where you don't necessarily expect to find it.

To describe our little programming language, fix an ordered list

$$F = (a_1/b_1, a_2/b_2, \dots, a_r/b_r)$$

of positive fractions (all in lowest common terms). We want to think of F as a program that determines a partial function $\hat{F} : \mathbb{N}^k \nrightarrow \mathbb{N}$.

First, given a natural number x, define $F(x)$ as follows:

- $x\,\frac{a_i}{b_i}$ where i is minimal such that $b_i \mid x$.
- x, if there is no such i.

In other words, we find the first fraction in our list whose denominator divides x, and then multiply x by that fraction; essentially, we replace a factor b_i in x by a_i. If there is no suitable fraction we simply return x as a default value.

This procedure describes one step in a computation, for the whole computation we apply F repeatedly until a fixed point is reached; if that never happens, $\widehat{F}$ is undefined on x.

We still need some input/output convention. There are several options, here is a fairly natural one. We refer to these programs as normal. Using the standard prime encoding, we can write input $e_1, e_2, \dots, e_k$ as

$$x = 3^{e_1} 5^{e_2} \dots p_k^{e_k}$$

Now suppose $F^t(x) = 2^e$ is a fixed point (so 2 is not a denominator); then we let

$$\widehat{F}(e_1, \ldots, e_k) = e$$

Again, if there is no fixed point of this form, we think of the function as being undefined. As already mentioned, in general, FRC programs do not produce total functions.

Example 1: We want to compute addition, in the form of $3^a 5^b \rightsquigarrow 2^{a+b}$. The following program works:

$$F = (2/3, 2/5)$$

Note the order of the fractions does not matter in this case, we are just moving pebbles from one place to another.

Example 2: We want to compute proper subtraction, in the form of $3^a 5^b \rightsquigarrow 2^{\max(0,a-b)}$ The following program works:

$$F = (1/15, 2/3, 1/5)$$

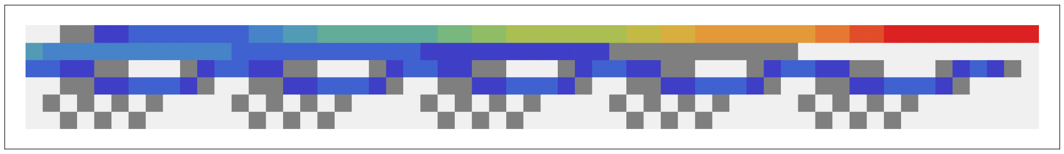
This time, the order of the fractions does matter. E.g., for $a \geq b$ we have

$$3^a 5^b \rightsquigarrow 3^{a-b} 5^0 \rightsquigarrow 2^{a-b}$$

Example 3: We want to compute multipication, in the form of $3^a 5^b \rightsquigarrow 2^{a \cdot b}$. The following program works:

$$F = \left(\frac{182}{55}, \frac{11}{13}, \frac{1}{11}, \frac{5}{7}, \frac{11}{3}, \frac{1}{5}\right)$$

Note the use of extra prime factors in this case. Here is the computation of $5 \times 3 = 15$. In the figure, the rows correspond to the primes used in the program, here 2, 3, 5, 7, 11 and 13, from top to bottom. Time flows from left to right. Instead of writing down the exponent of the prime decomposition of the current value of x, we use colors (trust me, nobody wants to read tables of boring numbers). The redder the color, the larger the number.



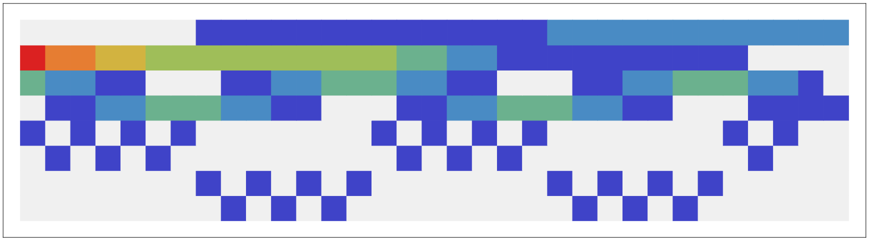
Example 4: We want to compute quotient and remainder simultaneously, so we adjust our I/O convention to

$$3^a 5^b 11 \rightsquigarrow 2^{\,a \operatorname{div} b}\, 7^{\,a \bmod b}$$

The following program works:

$$F = \left(\frac{91}{165}, \frac{11}{13}, \frac{1}{55}, \frac{34}{11}, \frac{95}{119}, \frac{17}{19}, \frac{11}{17}, \frac{1}{5}\right)$$

Here is a picture of the computation for $a = 7$ and $b = 3$, producing $2^2 7^1$:



With a bit of effort one can actually figure out the numbers hiding behind the colors.

Example 5:

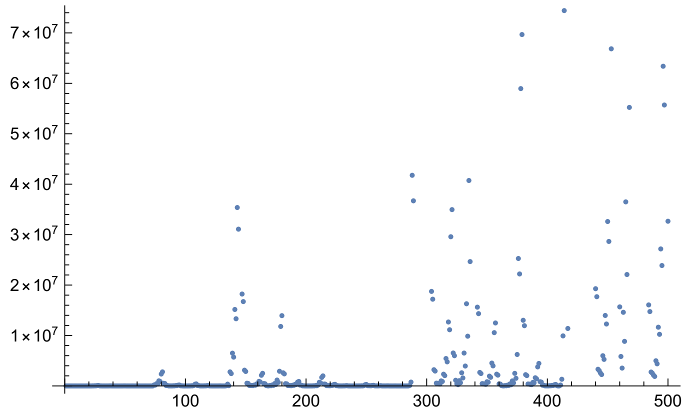
Here is a rather more sophisticated and exceedingly opaque program due to Conway (you thought the obfuscated C code competition was bad?):

$$\left(\frac{17}{91}, \frac{78}{85}, \frac{19}{51}, \frac{23}{38}, \frac{29}{33}, \frac{77}{29}, \frac{95}{23}, \frac{77}{19}, \frac{1}{17}, \frac{11}{13}, \frac{13}{11}, \frac{15}{2}, \frac{1}{7}, \frac{55}{1}\right)$$

Note the last term: this program will not terminate, it produces an infinite sequence of intermediate values $F^t(x)$ that we interpret as its output (actually, we will filter out the values we really care about; see below). Here is the behavior on input $x = 2$:

$$\begin{gathered}2, 15, 825, 725, 1925, 2275, 425, 390, 330, 290, 770, 910, 170, 156, 132,\\ 116, 308, 364, 68, 4, 30, 225, 12375, 10875, 28875, 25375, 67375, 79625,\\ 14875, 13650, 2550, 2340, 1980, 1740, 4620, 4060, 10780, 12740, 2380,\\ 2184, 408, 152, 92, 380, 230, 950, 575, 2375, 9625, 11375, 2125, 1950,\\ 1650, 1450, 3850, 4550, 850, 780, 660, 580, 1540, 1820, 340, 312, 264, \ldots\end{gathered}$$

Looks like so much gibberish. A plot for these values during the first 500 steps of the program run is no more illuminating.



This looks rather chaotic and, frankly, incomprehensible, until one remembers our original output convention: we should be looking for powers of 2. As soon as we filter out these, the fog lifts:

$$2^1, 2^2, 2^3, 2^5, 2^7, 2^{11}, \ldots, 2^{15485863}, \ldots$$

Conways program enumerates the prime numbers—except for a little glitch at 2^2. Quite amazing. The glitch could be fixed, but it would ruin the beautiful simplicity of our program.

2 Universality of FRC

With a little more effort one can convince oneself that FRC-programs can handle arithmetic, and control flow. So it may not be a total surprise that one can estable the following theorem.

Theorem 2.1 *Every partial computable function can be computed by a* FRC *program.*

Thus, FRC-computability is equivalent to any other model of computation that you will encounter later (Turing machines, register machines, the λ-calculus, and so on).

At any rate, To prove this theorem, we have to do two things:

- pick any of the standard models of computation, and
- show that the model can be "simulated" by a FRC program.

Needless to say, it is critical to pick a convenient reference model: the standard models that you will encounter in 15-251 (Turing machines [3], Herbrand-Gödel or the λ-calculus) do not work very well for this particular argument. A better starting point is a model that looks a bit more like a digital computer (well, more than all the other models). A register machine [2] consists of a number of registers, each holding a natural number, and has only three kinds of instructions (actually, 2.5).

• `inc r k`	increment register R_r, goto k.
• `dec r k l`	if $R_r > 0$, decrement register R_r and goto k, otherwise goto l.
• `halt`	well ...

That's it. A register machine program is a sequence of instructions $P = I_0, I_1, \ldots, I_{n-1}$ and is executed in the obvious way. We can choose particular registers for input and output.

For example, the following program performs addition:

```
// addition   R0 R1 --> R2
  0:   dec 0  1  2
  1:   inc 2  0
  2:   dec 1  3  4
  3:   inc 2  2
  4:   halt
```

This may seem rather primitive, but one can show without too much pain that RMs can simulate Turing machines, using the standard coding mumbo-jumbo. So, it suffices to simulate RMs via FRC-programs. This is still mildly tedious, but not hopeless, see the exercises.

Conway's theorem is surprising since the integers tend to become computationally difficult only when one deals with addition and multiplication at the same time. One can show that addition-only (Presburger arithmetic) and multiplication-only (Skolem arithmetic) are both computationally simple. Surprisingly, Conway's result does not use addition; multiplication alone here already makes a mess (this is a white lie, why?).

In general, trouble starts as soon as both operations together are available. The classical result here is a famous theorem.

Theorem 2.2 (Davis, Putnam, Robinson, Matiyasevic) *It is undecidable whether a polynomial equation with integer coefficients*

$$P(x_1, x_2, \dots, x_n) = 0$$

has a solution in the integers.

The solution must be integral, not real or complex (somewhat counterintuitively, these are much easier to handle). Hilbert posed this problem in 1900, it took 70 years to prove the theorem.

---

3 Exercises

Exercise 3.1 Determine how the division FRC-program works and prove that it is correct.

Exercise 3.2 Determine the running time of the quotient/remainder program.

Exercise 3.3 Produce FRC programs for other arithmetic functions such as factorial and exponential.

Exercise 3.4 Try to prove that Conway's prime-generating program is correct.

Exercise 3.5 Figure out how to implement register machine instructions in a FRC-program. Then combine them to get a FRC program that simulates a given register machine.

References

[1] J. H. Conway. FRACTRAN: a simple universal programming language for arithmetic. In J. C. Lagarias, editor, *The ultimate challenge. The* $3x + 1$ *problem*, pages 249–264. Providence, RI: American Mathematical Society (AMS), 2010.

[2] J. C. Shepherdson and H. E. Sturgis. Computability of recursive functions. *JACM*, 10:217–255, 1963.

[3] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *P. Lond. Math. Soc.*, 42:230–65, 1936.