Perfect Prime Predictors

by Sven Nilsen, 2017

Abstract: Perfect prime predictors belong to 4 kinds of function spaces, and the constraints of these function spaces tell us that there is ***likely no trivial way to find primes efficiently****, because all perfect prime predictors either ignore the distribution of primes or reflects it.* ***Yet, if there is, I suggest a way to find it faster*** *than checking every perfect prime predictor.*

Path semantics is an extension of type theory that grounds meaning (semantics) using functions, started in 2014 by the author. A “path” is a function that predicts something about another function, which are classified into “symmetric” and “asymmetric” paths. A generic symmetric path is written `f[g]` where `f` and `g` are functions. A generic asymmetric path is written `$f[g_{i \to n}]$` where `$g_{i \to n}$` is a Nilsen cartesian product of the form `$g_0 \times g_1 \times \ldots \times g_{n-1} \to g_n$`, a kind of type calculus where types are replaced by functions. Asymmetric paths are a generalization of symmetric paths, so one can convert easily to asymmetric form when necessary.

A symmetric path has the axiom:

f[g](g(x)) ?= g(f(x))

It says that for a function `f` and a property defined by `g`, there sometimes exists a function `f[g]` that predicts the property of the output for function `f`.

There is no known efficient and fully automated algorithm to find paths, so a lot of ingenuity and creativity is applied that cross pollinates ideas with the field of machine learning and artificial intelligence. It turns out that path semantics also is very suitable for theorem proving and is closely related to dependently types and creating smarter interactive proof assistants.

A perfect prime predictor is a function that takes a natural number and outputs a natural number, but has a symmetric path to a boolean function which truth values determine whether a number is a prime. Therefore, path semantics is a natural tool to study perfect prime predictors.

f: nat → nat

f[is_prime]: bool → bool

is_prime: nat → bool

The number of boolean functions of `n` input arguments is:

$$2^{2^n}$$

Since `2^(2^1) = 4`, there are only 4 logically equivalent functions of the type `bool → bool`. This means that every perfect prime predictor falls into one of 4 kinds of function spaces.

Boolean functions can be enumerated using software library for discrete combinatorics, for example the Rust library “discrete” on http://crates.io. This library uses phantom types to derive numerical algorithms that maps to and from the natural numbers. One can use the type `PowerSet<Of<DimensionN>>>` for any boolean function.

Here is a program that enumerates all functions of `bool → bool` and prints out their sets:

```
extern crate discrete;

use discrete::*;

fn main() {
        let space: PowerSet<Of<DimensionN>> = Construct::new();
        let ref dim = vec![2];
        let ref mut pos = vec![];
        let count = space.count(dim);
        println!("Count: {}", count);
        for ind in 0..count {
                space.to_pos(dim, ind, pos);
                println!("{:?}", pos);
        }
}
```

The output of the program is interpreted as for which input the function returns `true`:

Output	**Name**	**Dyon closure**
[]	false_1	\(_) = false
[[0]]	not	\(x) = !x
[[1]]	id	\(x) = x
[[0], [1]]	true_1	\(_) = true

Examples of perfect prime predictors:

f[is_prime] <=> false_1

• mul(2)	Anything multiplied with 2 is never a prime
• \(_) = 4	4 is not a prime
• \(x) = 4+2*x	Even numbers starting from 4 are never primes

f[is_prime] <=> not

• \(x) = if is_prime(x) {1} else {3}	1 is not a prime, 3 is a prime

f[is_prime] <=> id

• id	Obviously the output is a prime depending on the input

f[is_prime] <=> true_1

• prime_n	Finds the nth prime using a prime sieve algorithm
• \(_) = 3	3 is a prime

One can see from the examples of perfect prime predictors, that not every one of them is useful to find primes. In fact, most perfect prime predictors are absolutely useless. However, if there is a trivial and efficient way to find primes perfectly, then it must be a perfect prime predictor.

Since it is known that every perfect prime predictor are one of 4 kinds, it is possible to construct algorithms for enumerating all perfect prime predictors of each kind up to number N. I will not create such programs, but rather show the patterns.

Basically, you create two lists, `a` (primes) and `b` (non-primes) and pick one from each list depending on the back-projected pattern from the functions of type `bool → bool` to the functions of type `nat → nat`. I changed the background color to make it easier to see the pattern:

Input	false_1	not	id	true_1
0	b	a	b	a
1	b	a	b	a
2	b	b	a	a
3	b	b	a	a
4	b	a	b	a
5	b	b	a	a
6	b	a	b	a
7	b	b	a	a
8	b	a	b	a
9	b	a	b	a
10	b	a	b	a
11	b	b	a	a

For example, if you enumerate functions of `f[is_prime] <=> false_1` and `f[is_prime] <=> id`, you will get output similar to this (each row is a function):

f[is_prime] <=> false_1

```
0    1    2    3    4
0    0    0    0    0
1    0    0    0    0
4    0    0    0    0
0    1    0    0    0
1    1    0    0    0
4    1    0    0    0
0    4    0    0    0
1    4    0    0    0
4    4    0    0    0
0    0    1    0    0
1    0    1    0    0
4    0    1    0    0
0    1    1    0    0
1    1    1    0    0
4    1    1    0    0
0    4    1    0    0
1    4    1    0    0
4    4    1    0    0
0    0    0    1    0
1    0    0    1    0
4    0    0    1    0
0    1    0    1    0
1    1    0    1    0
4    1    0    1    0
0    4    0    1    0
1    4    0    1    0
4    4    0    1    0
0    0    1    1    0
…
4    4    4    4    4
```

f[is_prime] <=> id

```
0    1    2    3    4
0    0    2    2    0
1    0    2    2    0
4    0    2    2    0
0    1    2    2    0
1    1    2    2    0
4    1    2    2    0
0    4    2    2    0
1    4    2    2    0
4    4    2    2    0
0    0    3    2    0
1    0    3    2    0
4    0    3    2    0
0    1    3    2    0
1    1    3    2    0
4    1    3    2    0
0    4    3    2    0
1    4    3    2    0
4    4    3    2    0
0    0    2    3    0
1    0    2    3    0
4    0    2    3    0
0    1    2    3    0
1    1    2    3    0
4    1    2    3    0
0    4    2    3    0
1    4    2    3    0
4    4    2    3    0
0    0    3    3    0
...
4    4    3    3    4
```

A prime sieve algorithm, or a potential trivial algorithm for mapping natural numbers to primes, will output only primes. It completely ignores the primality of the input:

0	1	2	3	4	5	6	7	8	9	10	11
2	3	5	7	11	13	17	19	23	29	31	37

One can also imagine a function that only outputs non-primes for primes and primes for non-primes the following way:

0	1	2	3	4	5	6	7	8	9	10	11
2	3	0	1	5	4	7	6	11	13	17	18

This means that all perfect prime predictors either completely ignores the distribution of primes, or they reflect the distribution "secretly" in the output from the input.

The consequence is that no matter how many perfect prime predictors that are checked to learn something new about the distribution of primes, there is really no trivial way to extract the knowledge, because in order to interpret the output, one would already need to know the distribution of primes!

However, if there is a magical efficient perfect prime predictor, then one can exploit the secret pattern in the following way:

1. Function composition is restricted to those who "adds up to" the pattern
2. Execution branching is restricted to those who "adds up to" the pattern

This means that every way of picking or deciding under what conditions to evaluate functions, leads to 4 kinds of proof goals, each for finding the puzzle piece that corrects the pattern back such that the function becomes a perfect prime predictor. *If an algorithm was developed to do this efficiently, it can also be used for any other perfect prediction problem, since back projection is universal.*

To explain this idea intuitively, a function `a` that outputs the primes up some limit:

0	1	2	3	4	5	6	7	8	9	10	11
2	3	5	7	11	13	0	0	0	0	0	0

Can be combined by addition with a function `b` that outputs the rest of the primes:

0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	17	19	23	29	31	37

Why these two functions should be combined, is part of the proof goal:

```
(\(x) = a(x) + b(x))[is_prime] <=> true_1
```

This might sound like a very inefficient search, because the choice of any function, changes which other functions you might consider as candidates for correcting the pattern. It is surely non-trivial. However, in path semantics there is a theorem that you only need 2 objects to disprove the existence of paths. If they are both primes or non-primes, but the output of the 2 objects differs in primality, then the path does not exist. The efficiency of the search strongly depends on how quickly one can find 2 such objects, and if you can not find them, it leaves the particular combination of those functions as a potential candidate for a perfect prime predictor.