An Exploration of the Formal Properties of PromQL

Brian Brazil
Founder

Robust Perception

Who am I?

A Computer Scientist passionate about expanding the field of CS, with a particular interest on the question of the power of operational monitoring.

Winner of the Victor W. Graham prize for Pure Mathematics, and Scholar of Computer Science in Trinity College Dublin.

Google SRE for 7 years, working on expanding the horizons of what was considered possible with monitoring tools.

I also contribute to some obscure monitoring system, that'll only ever be of academic interest as it uses pull and we all know that can't scale.
You probably haven't heard of it.

Robust **Perception**

What is PromQL

Three parts:

- Data model
- Query language
- Execution environment

Data model

PromQL's data model is based on "labels" and "samples", combining into "time series".

A "time series" comprises a unique set of "labels" and one or more "samples".

"labels" are a set of pairs of strings, where the first of the pair is unique.

A sample is a pair of (int64, float64), within a "time series" the first of the pair is unique.

What can we describe with this model?

List

```
{item="1"}  (0, 7)

{item="2"}  (0, -Inf)

{item="3"}  (0, 3)
```

Grid

```
{x="1", y="1"}  (0, 13)

{x="2", y="1"}  (0, NaN)

{x="1", y="2"}  (0, 90)

{x="2", y="2"}  (0, .7)
```

Query language

Can select time series based on existence/non-existence of one or more of their labels. From there there's a few broad classes of operations and functions:

Relational algebra on labels across time series

Functions on the samples within a time series

Operations/Functions on single samples across time series

Type conversion functions (PromQL is strongly and statically typed)

Display related functions

...and one function to manipulate labels.

Label Manipulation

Normally you can only work with labels that are the same, e.g. `{x="1", y="1"}` and `{x="2", y="1"}` both have `y="1"`.

The `label_replace()` function allows us to change the `1` to a `2` via regular expressions.

Robust **Perception**

Execution environment

The environment has a counter, and when looking across time series only the samples with the first value between this counter and 300,000 less than it are available. Of these the sample with the highest first value will be available.

In each cycle this counter increases, typically by 1,000 to 60,000.

Output of expressions executed in a cycle can be added to the data set, with the first value of samples set to that of the counter.

Expressions are evaluated simultaneously, may see output of other expressions from this cycle.

So how powerful is this language?

If we can model a list, can do relational algebra and can record new results each cycle then we can use the list as a tape and build a Finite State Machine.

Been there, done that.

What else can we do that's as powerful, but a little more aesthetically pleasing?

Could build a grid and run a simple 2-dimensional cellular automata on top of it.

Automata

Go for something simple, only two states - 0 and 1.

Decimal would be hard for grid positions, so use unary numbering:
1 = 1, 2 = 11, 3 = 111, 4 = 1111 etc.

We'll have a time series called `{__name__="init"}` that's the size of the grid, and all with the value 1.

The grid will be stored in`{__name__="grid"}`

For the edges of the grid, we'll do simple manipulation of the counter value to generate random-ish values as there's no RNG available in PromQL.

If exactly three neighbours are 1, become 1

```
(
 (
  (
   (label_replace(grid, "x", "$1", "x", "^1(.*)$") or init * scalar(round((vector(time()) + 0) % 8 / 8)))
    + (label_replace(grid, "x", "1$1", "x", "^(.*)$") or init * scalar(round((vector(time()) + 1) % 8 / 8)))
    + (label_replace(grid, "y", "$1", "y", "^1(.*)$") or init * scalar(round((vector(time()) + 2) % 8 / 8)))
    + (label_replace(grid, "y", "1$1", "y", "^(.*)$") or init * scalar(round((vector(time()) + 3) % 8 / 8)))
    + (label_replace(label_replace(grid, "y", "$1", "y", "^1(.*)$"), "x", "$1", "x", "^1(.*)$")
          or init * scalar(round((vector(time()) + 4) % 8 / 8)))
    + (label_replace(label_replace(grid, "y", "$1", "y", "^1(.*)$"), "x", "1$1", "x", "^(.*)$")
          or init * scalar(round((vector(time()) + 5) % 8 / 8)))
    + (label_replace(label_replace(grid, "y", "1$1", "y", "^(.*)$"), "x", "$1", "x", "^1(.*)$")
          or init * scalar(round((vector(time()) + 6) % 8 / 8)))
    + (label_replace(label_replace(grid, "y", "1$1", "y", "^(.*)$"), "x", "1$1", "x", "^(.*)$")
          or init * scalar(round((vector(time()) + 7) % 8 / 8)))
    ) == 3
  ) * 0 + 1
)
```

Robust **Perception**

Public Health Warning

If you feel uneasiness, headaches or existential dread as a result of this presentation stop watching immediately and consult your local Computer Numerologist for advice.

Do not look at the PromQL expressions in these slides with the remaining eye.

Do not attempt to create PromQL expressions of this complexity without professional advice.

If you discard this advice and summon an elder abomination from another realm, Robust Perception Ltd. accepts no responsibility for a) your meddling in higher mathematics or b) Charlie Stross having to throw away yet another book due to reality getting weirder than fiction.

Robust **Perception**

If you are 1 and exactly two neighbours are 1, stay 1

```
(
 (
  (
   (grid == 1) * 0
    + (label_replace(grid, "x", "$1", "x", "^1(.*)$") or init * scalar(round((vector(time()) + 0) % 8 / 8)))
    + (label_replace(grid, "x", "1$1", "x", "^(.*)$") or init * scalar(round((vector(time()) + 1) % 8 / 8)))
    + (label_replace(grid, "y", "$1", "y", "^1(.*)$") or init * scalar(round((vector(time()) + 2) % 8 / 8)))
    + (label_replace(grid, "y", "1$1", "y", "^(.*)$") or init * scalar(round((vector(time()) + 3) % 8 / 8)))
    + (label_replace(label_replace(grid, "y", "$1", "y", "^1(.*)$"), "x", "$1", "x", "^1(.*)$")
           or init * scalar(round((vector(time()) + 4) % 8 / 8)))
    + (label_replace(label_replace(grid, "y", "$1", "y", "^1(.*)$"), "x", "1$1", "x", "^(.*)$")
           or init * scalar(round((vector(time()) + 5) % 8 / 8)))
    + (label_replace(label_replace(grid, "y", "1$1", "y", "^(.*)$"), "x", "$1", "x", "^1(.*)$")
           or init * scalar(round((vector(time()) + 6) % 8 / 8)))
    + (label_replace(label_replace(grid, "y", "1$1", "y", "^(.*)$"), "x", "1$1", "x", "^(.*)$")
           or init * scalar(round((vector(time()) + 7) % 8 / 8)))
   ) == 2
 ) * 0 + 1
)
```

Robust Perception

Otherwise, become 0

```
init * 0
```

Joining these together

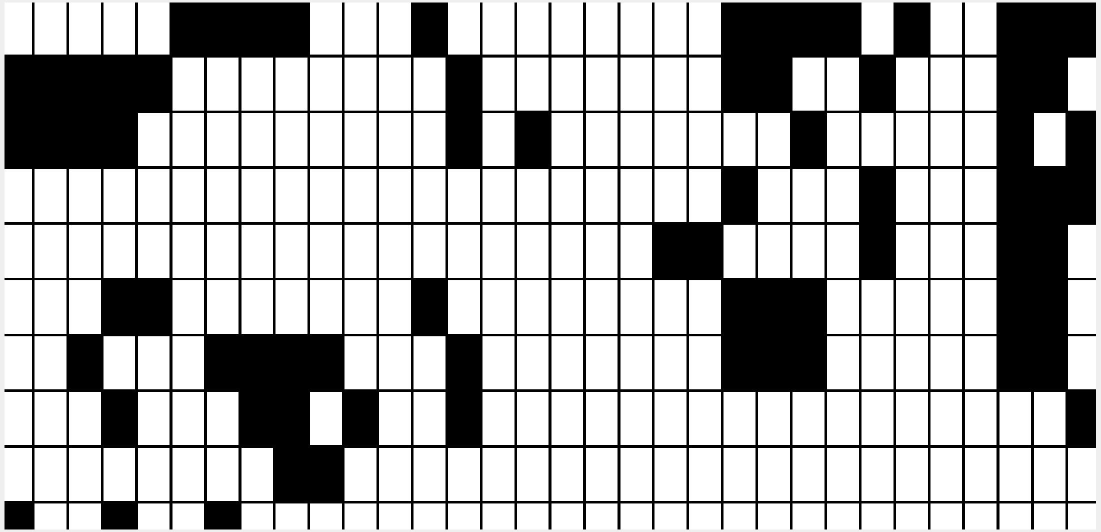
The `or` operator joins these together.

It'd be nice to have each in separate expressions and then do the `or`, but there's no guarantee which order the expressions run in.

And we would pick up data from the wrong cycle (have to always specify all points in the grid so everything is always overwritten).

What does this make?

What does this make?



Robust **Perception**

Conway's Life

A cellular automaton devised in 1970 by John Conway.

It is known to be Turing Capable, proved by Conway in 1982.

Full Turing Machines have been implemented by Paul Rendell (2000), Paul Chapman (2002), and Adam P. Goucher (2009).

This means that if it's possible to compute it on a computer, you can computer it in Conway's Life, and thus in PromQL.

No monitoring system can have a language more powerful than PromQL!

Resources

Blog Post: http://www.robustperception.io/conways-life-in-prometheus/

Demo: http://demo.robustperception.io:9090/consoles/life.html

Robust Perception Blog: www.robustperception.io/blog

Queries: prometheus@robustperception.io

Robust **Perception**