# Search-Based Regular Expression Inference on a GPU

Mojtaba Valizadeh    Martin Berger

https://martinfriedrichberger.net/

21 June 2023

# Problem

Can we accelerate program synthesis with GPUs?

# What makes program GPU-friendly?

- Predictable data movement
- Minimise data-dependent branching
- Maximise parallelism
- Minimise synchronisation between different threads/warps/cores etc as much as possible

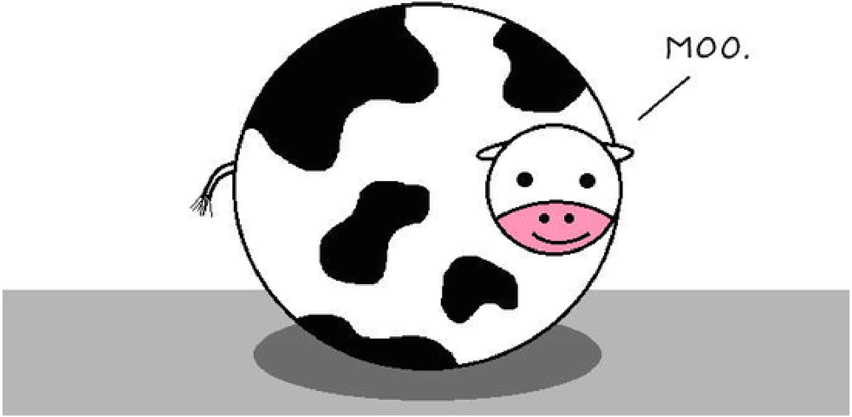
Can we do this for program synthesis?

# What makes program GPU-friendly?

- Predictable data movement
- Minimise data-dependent branching
- Maximise parallelism
- Minimise synchronisation between different threads/warps/cores etc as much as possible

Can we do this for program synthesis?

What is a tractable synthesis problem to bootstrap GPU synthesis?


MOO.

# Regular expression inference

# REI = programming by example with regular expressions

# REI = programming by example with regular expressions

**Input:**

- **Examples**: two finite sets of strings, $P$ and $N$
- **Cost function**: cost($\cdot$) for REs

**Output:** regular expression $r$ that is:

- **Precise**: $r$ accepts all strings in $P$ and rejects all strings in $N$
- **Minimal**: no regular expression with a cost less than cost($r$) is precise

# REI = programming by example with regular expression

Old & well-known problem (1967 Gold's "Language identification in the limit")

Currently unsolved for deep-learning based

REs are "embarrassingly sequential"

Asymptotic complexity well understood

- Gold (1978): is NP-hard for DFAs
- Pitt and Warmuth (1993): is NP-hard for DFAs, NFAs and REs, even to approximate minimum

# Regular expressions

The **regular expressions** over $\Sigma$ are given by the following grammar:

$$r \quad ::= \quad \emptyset \mid \epsilon \mid a \mid r \cdot r \mid r + r \mid r^*$$

We write Lang($r$) for the language of $r$

Usual abbreviations, e.g., $rr'$ for concatenation $r \cdot r'$, $r?$, $\Sigma^*$. Also implemented: intersection, negation, complement

# Search order, cost homomorphism

A **cost function** is a map $\text{cost}(\cdot) : \text{RE}(\Sigma) \to \text{Nat}$. It is a **cost homomorphism** if there are integer constants, $c_1, ..., c_5 > 0$, such that:

- $\text{cost}(\emptyset) = \text{cost}(\epsilon) = \text{cost}(a) = c_1$ for all $a \in \Sigma$
- $\text{cost}(r?) = \text{cost}(r) + c_2$
- $\text{cost}(r^*) = \text{cost}(r) + c_3$
- $\text{cost}(r \cdot r') = \text{cost}(r) + \text{cost}(r') + c_4$
- $\text{cost}(r + r') = \text{cost}(r) + \text{cost}(r') + c_5$

We call each $c_i$ the **cost** of the corresponding regular constructor

# Solution: trivial algorithm

Enumerate by increasing cost and check each if meets constraints

# Solution: trivial algorithm

Enumerate by increasing cost and check each if meets constraints

**Research question:** How to make GPU friendly?

# Core problem: how to represent REs during search?

Using REs search space is wasteful for several reasons:

- **Not space-efficient:** Each regular expression is a tree (needs pointers)
- **Slow contains-check:** To check candidate, needs tree-walk
- **Redundant:** Each regular language is denoted by infinitely many regular expressions. For example, 00+1 and 1+00 denote the same language
- **Non-local:** Pointers can point to anywhere in memory, unpredictable

# Representation of RE by languages

Recall language $L$ is subset of $\Sigma^*$. Isomorphic representation as **characteristic sequence**

$$\begin{aligned} \mathbf{1}_L : \quad \Sigma^* &\to \mathbb{B} \\ \sigma &\mapsto \begin{cases} 1 & \sigma \in L \\ 0 & \text{else} \end{cases} \end{aligned}$$

Note: $\mathbb{B}$ is a semiring, and boolean algebra

Many REs have the same characteristic sequence e.g. $r + r$ and $r$. Characteristic sequence prune away this redundancy (REs up to observational congruence)

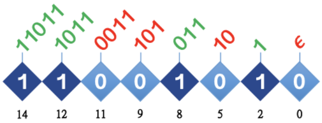
Problem: the domain $\Sigma^*$ of $\mathbf{1}_L$ is infinite

# Representation of languages

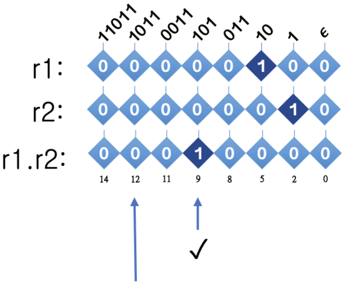
In REI, we only care about strings in $P \cup N$, we could use

$$\mathbf{1}_L : \quad P \cup N \quad \to \quad \mathbb{B}$$

Positive = {1, 011, 1011, 11011}
Negative = {$\epsilon$, 10, 101, 0011}



Note: $\mathbf{1}_L : P \cup N \to \mathbb{B}$ is not enough, e.g. with




1011 = "".1011 checked

= 1.011 checked

= 10.11 ⁉

= 101.1

= 1011.""

$P = 1, 001, 1001, 11011$

$N = \epsilon, 10, 101, 0011$

# Infix-closure

Our representation of regular expressions are characteristic sequences

$$cs : \mathrm{ic}(P \cup N) \to \mathbb{B}$$

$w$ is an **infix** (aka substring) of string $\sigma$ if $\sigma = w_1 \cdot w \cdot w_2$. Example: “abc” has infixes like

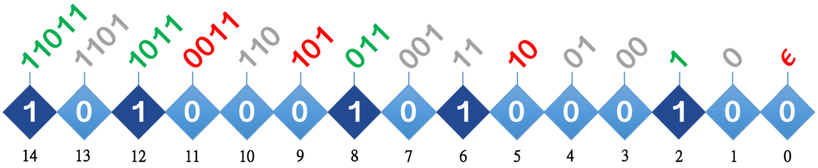
$$abc \quad ab \quad bc \quad a \quad b \quad c \quad \epsilon$$

ic($S$), the **infix-closure** of a set $S$, is the smallest infix-closed superset of $S$

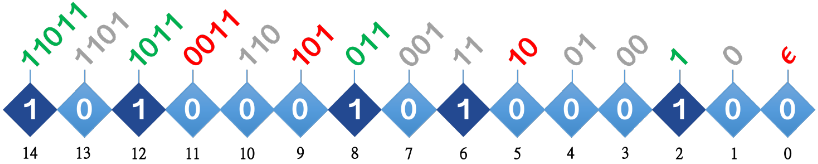
# Infix-closure: example

$P = \{1, 011, 1011, 11011\}$ and $N = \{\epsilon, 10, 101, 0011\}$. Then ic$(P \cup N)$ is

$$11011, 1101, 110, 11, 1011, 101, 10, 1, 011, 01, 0011, 001, 00, 0, \epsilon$$



# Infix-closure characteristic sequences summary




- Characteristic sequence turns language into a bitvector in memory
- bitvector = integer, bitvectors = integer matrix!
- *P* and *N* are fixed, so don't change during an REI run!
- ic($P \cup N$) is ordered in memory
- Going through ic($P \cup N$) is a linear scan (= predictable)

# Principled program synthesis: using formal power series over semirings

Our characteristic sequences $cs : \mathrm{ic}(P \cup N) \to \mathbb{B}$ form a ***-semiring**, abstracting:

- addition
- multiplication
- iteration

corresponding exactly to the operations on regular expressions

**Slogan:** *-semiring as the API for regular expression synthesis

**Algorithm 1** Main function of synthesis algorithm

**Input** Positive and negative examples $(\mathcal{P}, \mathcal{N})$, $cost$, $maxCost$
**Output** A minimal RE w.r.t. cost and $maxCost$ and consistent with $(\mathcal{P}, \mathcal{N})$, otherwise "not_found"

---

**if** $\mathcal{P} == \{\}$ **then return** $\emptyset$
**if** $\mathcal{P} == \{""\}$ **then return** $\epsilon$
$languageCache$ = [list of CSs of alphabet] ▷ $languageCache$ is global variable
**for** $c \leftarrow cost(\epsilon) + 1$ to $maxCost$ **do**
    $questions$ = buildQuestionMark( $c$ - $cost(?)$)
    $stars$ = buildStar$(c - cost(*))$
    $concats$ = buildConcat$(c - cost(\cdot))$
    $unions$ = buildUnion$(c - cost(+))$
    $languageCache[c]$ = questions ++ stars ++ concats ++ unions ▷ ++ is concatenation
**return** "not_found" ▷ Procedures in loop will return solution directly to caller of main, if found

# Language Cache

We construct all needed languages bottom-up, from lower to higher cost, and keep the constructed languages in memory for later re-use:

| | 11011 | 1101 | 1011 | 0011 | 110 | 101 | 011 | 001 | 11 | 10 | 01 | 00 | 1 | 0 | $\epsilon$ | cost |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $0$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| $1$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| $0?$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 2 |
| $01$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 3 |
| $0 + 1$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 3 |
| $(0 + 1)^*$ | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 4 |
| $0?\,1$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 4 |
| $(0?\,1)^*$ | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 5 |
| $(0 + 1)^*1$ | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 6 |
| ✓ $(0?\,1)^*1$ | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 7 |
| | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |

**Algorithm 2** Pseudocode for concatenation (*buildConcat* procedure in Alg. 1)

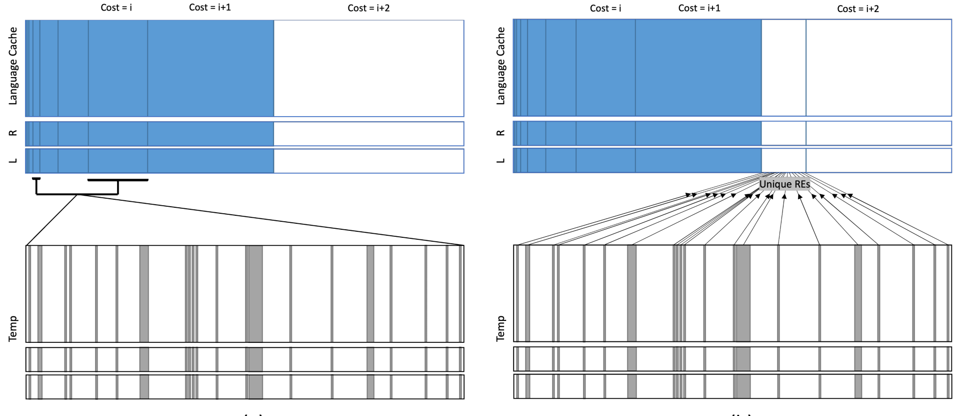
```
Input cost c, globals used: languageCache, P, N
Output A list of new CSs generated by concatenation
________________________________________________
outList ← []
for all L, R such that L + R = c do
    for all lCS ∈ languageCache(L) do
        for all rCS ∈ languageCache(R) do
            i ← 1
            newCS ← 0
            for w ← 0 to #ic(P ∪ N) − 1 do
                for all pair (l, r) ∈ gt[w] do
                    if (lCS & l) ≠ 0 and (rCS & r) ≠ 0 then
                        newCS ← newCS | i
                i ← i ≪ 1
            isUnique ← hashSet.insert(newCS)
            if isUnique then
                if newCS ⊨ (P, N) then
                    print newCS and terminate program
                outList.insert(newCS)
return outList
```

All local, except global uniqueness check

In order to maximise parallelism on GPUs, we allocate each new characteristic sequence (which may be redundant) into a temporary array

Each new characteristic sequence is composed from older entries in the language cache by simple, fast and **local** bitvector operations

For **each** new characteristic sequence we run a **global** uniqueness check



# Measurements

AlphaRegex benchmarks (previously state-of-the-art) are too small.

Our new, large benchmarks suite

- Alphabet $\Sigma$,
- $le$ is the maximal length of example strings,
- $p$ and $n$, the numbers of positive and negative examples, respectively.

With those parameters, we define two complementary benchmark generation schemes. Both create instances $(P, N)$ by sampling uniformly from two different spaces of random strings.

- Type 1: $\{(P, N) \in \Sigma^{\leq le} \times \Sigma^{\leq le} \mid \forall w \in P \cup N. \#P = p, \#N = n, P \cap N = \emptyset\}$
- Type 2: $\{((P_0, ..., P_{le}), (N_0, ..., N_{le})) \in Y \times Y \mid \Sigma_i \#P_i = p, \Sigma_i \#N_i = n, \forall i. P_i \cap N_i = \emptyset\}$

| Input | | | | | CPU | GPU | | |
|---|---|---|---|---|---|---|---|---|
| Type | No | # P | # N | Cost Function | Sec | Sec | Speed-up | # REs |
| 1 | 50 | 10 | 12 | (1, 1, 1, 1, 1) | 5080.7850 | 4.9512 | **1026x** | 26,774,099,142 |
| 1 | 51 | 12 | 9 | (10, 1, 1, 1, 1) | 4699.8137 | 4.4966 | **1045x** | 23,824,118,297 |
| 1 | 73 | 10 | 11 | (1, 10, 1, 1, 1) | 5805.2168 | 3.7144 | **1562x** | 22,703,639,676 |
| 1 | 20 | 9 | 9 | (1, 1, 10, 1, 1) | 2893.4835 | 2.8935 | **1000x** | 13,567,472,188 |
| 1 | 73 | 10 | 11 | (1, 1, 1, 10, 1) | 2901.9297 | 2.9504 | **983x** | 11,706,686,339 |
| 1 | 31 | 8 | 9 | (1, 1, 1, 1, 10) | 5856.6925 | 3.9973 | **1465x** | 14,210,157,835 |
| 1 | 57 | 12 | 10 | (10, 10, 10, 10, 1) | 2804.6793 | 3.4322 | **817x** | 14,163,906,090 |
| 1 | 50 | 10 | 12 | (10, 10, 10, 1, 10) | 4519.9456 | 4.9096 | **920x** | 23,349,552,935 |
| 1 | 57 | 12 | 10 | (10, 10, 1, 10, 10) | 4301.8548 | 4.5243 | **950x** | 20,257,045,497 |
| 1 | 97 | 12 | 12 | (10, 1, 10, 10, 10) | 5608.7286 | 4.7782 | **1173x** | 19,680,542,658 |
| 1 | 61 | 12 | 10 | (1, 10, 10, 10, 10) | 2915.0938 | 3.0532 | **954x** | 14,322,039,866 |
| 1 | 88 | 12 | 9 | (20, 20, 20, 5, 30) | 6899.0045 | 4.6904 | **1470x** | 25,193,577,825 |
| 2 | 88 | 14 | 8 | (1, 1, 1, 1, 1) | 3783.9772 | 4.2462 | **891x** | 23,697,549,545 |
| 2 | 150 | 14 | 12 | (10, 1, 1, 1, 1) | 4228.2773 | 4.4120 | **958x** | 23,125,803,623 |
| 2 | 158 | 12 | 14 | (1, 10, 1, 1, 1) | 2975.9956 | 2.4887 | **1195x** | 11,432,891,412 |
| 2 | 136 | 11 | 14 | (1, 1, 10, 1, 1) | 3374.8873 | 3.6080 | **935x** | 18,241,755,827 |
| 2 | 107 | 12 | 12 | (1, 1, 1, 10, 1) | 2432.4320 | 4.4120 | **551x** | 24,954,272,802 |
| 2 | 32 | 10 | 7 | (1, 1, 1, 1, 10) | 7400.8135 | 4.6482 | **1592x** | 16,729,795,052 |
| 2 | 136 | 11 | 14 | (10, 10, 10, 10, 1) | 2907.9182 | 3.9689 | **732x** | 17,476,988,322 |
| 2 | 200 | 13 | 8 | (10, 10, 10, 1, 10) | 9687.7952 | 4.5366 | **2135x** | 6,037,014,423 |
| 2 | 107 | 12 | 12 | (10, 10, 1, 10, 10) | 3383.1937 | 4.5071 | **750x** | 20,697,274,025 |
| 2 | 81 | 8 | 14 | (10, 1, 10, 10, 10) | 3497.9013 | 4.6699 | **749x** | 21,869,903,022 |
| 2 | 88 | 14 | 8 | (1, 10, 10, 10, 10) | 3405.5536 | 4.1602 | **818x** | 21,889,508,744 |
| 2 | 158 | 12 | 14 | (20, 20, 20, 5, 30) | 5804.8112 | 4.9228 | **1179x** | 23,163,079,580 |
| **Average** | | | | | 4465.4493 | 4.1238 | **1077x** | 19,127,861,447 |

5160 benchmarks (200 Type 1, 230 Type 2, 12 cost functions). Examples hardest within timeout < 5 sec on GPU. GPU: Nvidia A100-SXM4, CPU: Intel Xeon, 2.20 GHz, 25 GB

# Future work

Algorithmic engineering, is uniqueness check for every new characteristic sequence optimal?

Context-free inference? (implemented on CPU, in progress: GPU)

Scaling: approximate REI (in progress)

Mathematics of synthesis, e.g.

$$\frac{\text{Regular grammar}}{\text{Infix-closure}} = \frac{\text{General grammar}}{\text{???-closure}}$$

Comparison with LLMs (paper under anonymous review)

# Thanks

Code: `https://github.com/MojtabaValizadeh/paresy`
Paper: `https://arxiv.org/abs/2305.18575`

Code:

Paper: