算术化-frontend

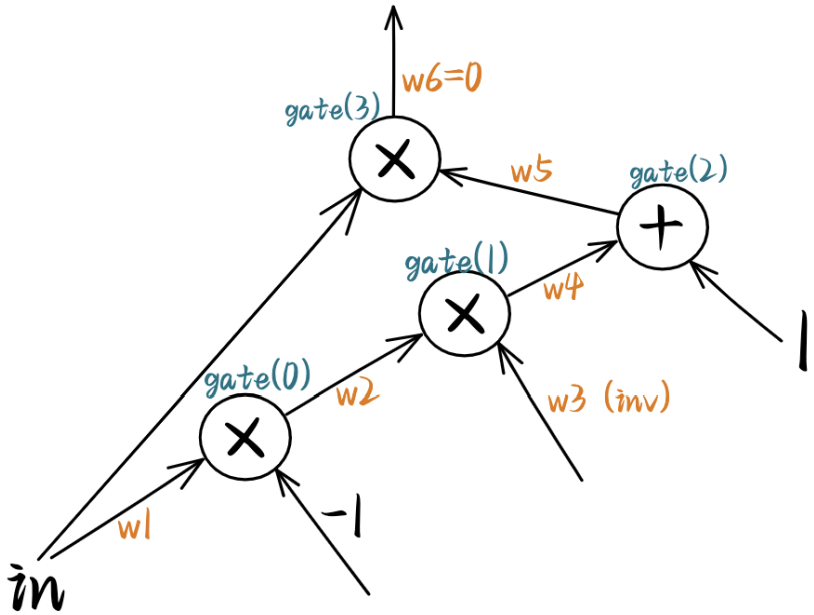
R1CS

首先来看一个 IsZero 电路如何转化为 R1CS 结构 :

> ### `IsZero` 电路
>
> Circom 为什么不能直接支持类似 `in==0` 的判断 ?
>
> 1. Circom 使用算术电路来构造零知识证明,每个门对应一条多项式计算。
> 2. 相等判断不满足电路门的输入输出需求,因此无法构造等式门。
> 3. 另外, 传统的编程语言支持的 `if/else` 分支,也不是电路门的形式,因此 Circom 不直接支持 if/else
> 4. 所以不能直接判断 `in==0` , 而要转换为 `inv` 的计算,来实现等价的多项式关系。
>
> Fortunately，Circom 有一个好用的库，包括一个 `IsZero` 返回 `0` 或 `1` 取决于输入是否为零的电路，如下所示：

```rust
template IsZero () {
    signal input in;
    signal output out;
    signal inv;

    inv <-- in!=0 ? 1/in : 0;  // 赋值, inv 即 inverse, 倒数
    out <-- -in*inv +1;    // 赋值
    out === -in*inv +1;    // 电路约束 gate0, gate1, gate2
    in*out === 0;          // 电路约束 gate3
}
```

> 请注意 `inv <-- in!=0 ? 1/in : 0` 这个表达式没有用 `==` 约束, 所以它不会被计入电路结构, 所以可以使用判等 , 可以使用三元运算符判断, 这里和普通的图灵完备编程语言没什么区别
>
> - if `in` 是 0 , 那么 $out = -in \times 0 + 1 = 1$
> - if `in` 是 1 , 那么 $out = -in \times \frac{1}{in} = 0$
> - ↑ 可以看到这行表达式可以让 `in * out` 始终 `== 0`
>
> 而 `out === -in*inv +1` 和 `in*out === 0` —— 这 2 行是电路结构 , Draw the arithmetic `circuit` representation of `IsZero` :
>
> 
>
>
> - w_i is `wire` (电线), $gate(i)$ 是电路门
> - in 是输入值, -1, 1 是输入常量
>
> 如图可见 `out === -in*inv +1;` 和 `in*out === 0` 这 2 个约束语句已经将电路描述出来了
>
> 以上代码可以进一步优化成 :

```
    inv <-- in!=0 ? 1/in : 0;
    out <== -in*inv +1;     // 优化: [赋值] 和 [约束] 合并
```

```
    in*out === 0;
```

看代码, The circom `IsZero` program can be "`flattened`" into **4** constraints, each of the form `left ◦ right = output` (flattened 拉平后, 每一步都能表示成 `Left * Right = Output` 的形式):

$$w_1 * (-1) = w_2 \tag{0}$$
$$w_2 * w_3 = w_4 \tag{1}$$
$$w_4 + 1 = w_5 \tag{2}$$
$$w_1 + w_5 = w_6 \tag{2}$$

The prover is claiming to know some **legal assignment** $\vec{x} = (x_1, x_2, x_3, x_4, x_5, x_6)$, so that when each value a_i is assigned to corresponding wire w_i, and w_6 = 0, the circuit is satisfied.

For each gate g_i , we create three wire vectors $\vec{l_i}$, $\vec{r_i}$, $\vec{o_i}$, containing the coefficients of each variable w_j at the gate. The wire vectors also include a constant term w_0 (常数项 w_0):

以 $Gate(0)$ 为例, 结合如上 circuit :
右输入乘的是个常数 -1 , 所以放在 w_0 位置 ;

$$g_0 : w_1 \cdot (-1) = w_2$$

gate(0)

门左输入 门右输入 门输出

$$\begin{array}{|c|ccccccc|} & w_0 & w_1 & w_2 & w_3 & w_4 & w_5 & w_6 \\ \vec{l}_0 = & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ \vec{r}_0 = & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \vec{o}_0 = & 0. & 0 & 1 & 0 & 0 & 0 & 0 \end{array}$$

平平无奇 $Gate(1)$

$$g_1 : w_2 \cdot w_3 = w_4$$

$$\begin{array}{|c|ccccccc|} & w_0 & w_1 & w_2 & w_3 & w_4 & w_5 & w_6 \\ \vec{l}_1 = & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ \vec{r}_1 = & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ \vec{o}_1 = & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{array},$$

加法门 $Gate(2)$, 加法门加的一般是常数或者已有变量, 所以一般不写在上述矩阵里, 如果要写的话, 就改写成乘法的形式 : $(w_4 + 1) \times 1$:

$$g_2 : \ w_4 + 1 = w_5$$
$$\implies (w_4 + 1) \cdot 1 = w_5$$

$$\begin{array}{|c|ccccccc|} & w_0 & w_1 & w_2 & w_3 & w_4 & w_5 & w_6 \\ \vec{l}_2 = & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ \vec{r}_2 = & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \vec{o}_2 = & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{array},$$

平平无奇 $Gate(3)$:

$$g_3 : w_1 \cdot w_5 = w_6$$

$$\begin{array}{|c|ccccccc|} & w_0 & w_1 & w_2 & w_3 & w_4 & w_5 & w_6 \\ \vec{l}_3 = & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ \vec{r}_3 = & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ \vec{o}_3 = & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}.$$

Now, we collect each of the left l_i wire vectors into a matrix

- $\mathcal{L} = \vec{l_0}, \vec{l_1}, \vec{l_2}, \vec{l_3}$, and likewise for the right
- $\mathcal{R} = \vec{r_0}, \vec{r_1}, \vec{r_2}, \vec{r_3}$, and output
- $\mathcal{O} = \vec{o_0}, \vec{o_1}, \vec{o_2}, \vec{o_3}$ vectors

$$\mathcal{L} = \begin{array}{c} \begin{matrix} w_0 & w_1 & w_2 & w_3 & w_4 & w_5 & w_6 \end{matrix} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array} \begin{matrix} \\ \vec{l}_0 \\ \vec{l}_1 \\ \vec{l}_2 \\ \vec{l}_3 \end{matrix}, \quad \mathcal{R} = \begin{array}{c} \begin{matrix} w_0 & w_1 & w_2 & w_3 & w_4 & w_5 & w_6 \end{matrix} \\ \begin{pmatrix} -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array} \begin{matrix} \\ \vec{r}_0 \\ \vec{r}_1 \\ \vec{r}_2 \\ \vec{r}_3 \end{matrix},$$

$$\mathcal{O} = \begin{array}{c} \begin{matrix} w_0 & w_1 & w_2 & w_3 & w_4 & w_5 & w_6 \end{matrix} \\ \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \begin{matrix} \\ \vec{o}_0 \\ \vec{o}_1 \\ \vec{o}_2 \\ \vec{o}_3 \end{matrix}.$$

The $\mathcal{L}, \mathcal{R}, \mathcal{O}$ matrices, along with our witness vector $\vec{x} = (x_1, x_2, x_3, x_4, x_5, x_6)$, gives the R1CS form of the IsZero circuit. A satisfying $\vec{x}$ fulfils the equation $\boxed{\mathcal{L} \cdot \vec{x} + \mathcal{R} \cdot \vec{x} = \mathcal{O} \cdot \vec{x}}$

这里的 $\vec{x}$ 一般被叫做 `Witness`

Quadratic Arithmetic Programs

> `QAP` equals to `R1CS`

The flow of QAP is **program → circuit → R1CS → QAP**

在下面示例中，我们将选择一个非常简单的问题：
求一个三次方程的解：`x**3 + x + 5 == 35` (hints: Answer is `3`)

```python
def qeval(x):    # python 代码表示
    y = x**3
    return x + y + 5
```

我们在这里使用的简单编程语言支持基本的算术 `(+、-、_、/)`、恒等幂指数 (x^7，但不是 x^y) 和变量赋值，这足够强大到理论上可以在其中进行任何计算 (只要计算步骤的数量是有界的且不允许循环)

请注意，不支持模 (%) 和比较运算符 `(<、>、≤、≥)`，因为在有限循环群算法中没有直接进行模 mod 或比较的有效方法（thanks to it；如果有办法做的话，椭圆曲线密码学被破解的速度比 “二分查找” 和 “中国剩余定理” 还要快）

1. Flatten

The first step is a “flattening” procedure, where we convert the original code, which may contain **arbitrarily** complex statements and expressions, into a sequence of statements that are of two forms:

1. `x = y` (where `y` can be a variable or a number) and
2. `x = y (op) z` (where `op` can be `+, -, *, /` and `y` and `z` can be variables, numbers or themselves sub-expressions).

You can think of each of these statements as being kind of like logic gates in a circuit. The result of the flattening process for the above code is as follows :

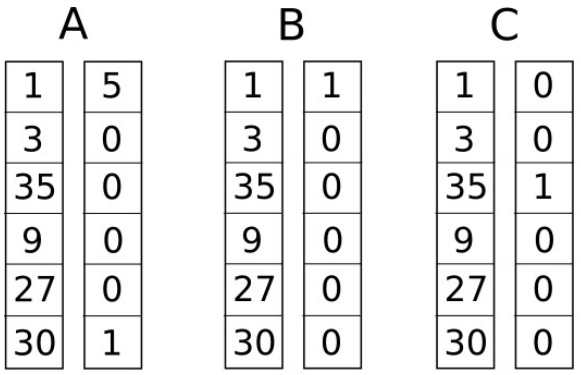
```
// x**3 + x + 5 == 35
sym_1 = x * x
y = sym_1 * x
sym_2 = y + x
~out = sym_2 + 5
```

2. Gates to R1CS

Now, we convert this into something called a rank-1 constraint system (R1CS). An R1CS is a sequence of groups of three vectors `(a, b, c)` , and the solution to an R1CS is a vector `s` , where `s` must satisfy the equation $s \cdot a \ * \ s \cdot b - s \cdot c = 0$ ($\cdot$ 是点乘)

例如，如下是一个令人满意的 `R1CS` ：

```
a = (5,0,0,0,0,1),
b = (1,0,0,0,0,0),
c = (0,0,1,0,0,0),
s = (1,3,35,9,27,30),
```



(注：第一个 `35 = 1*5 + 30*1`，第二个 `35 = 35 * 1`)

上述例子只是一个约束，接下来我们要将每个逻辑门（即 “压扁” 后的每一个声明语句）转化成一个约束（即一个 `(a, b, c)` 三向量组），转化的方法取决于声明是什么运算 `(+,-,*,/)` 和声明的参数是变量还是数字。

在我们这个例子中，除了“压扁”后的五个变量 (`'x', '~out', 'sym_1', 'y', 'sym_2'`) 外，还需要在第一个分量位置处引入一个冗余变量 `~one` 来表示数字 `1` ，就我们这个系统而言，一个向量所对应的 `6` 个分量是 (可以是其他顺序，只要对应起来即可):

```
'~one', 'x', '~out', 'sym_1', 'y', 'sym_2'
```

`Gate 1` :

```
sym_1 = x * x, 即 x*x - sym_1 = 0
```

我们可以得到如下向量组:

```
a = [0, 1, 0, 0, 0, 0]
b = [0, 1, 0, 0, 0, 0]
c = [0, 0, 0, 1, 0, 0]
```

如果解向量 `s` 的第二个标量是 3，第四个标量是 9，无论其他标量是多少，都成立，因为：a = 3 *1, b = 3* 1, c = 9 *1，即 `*a b = c 。 同样，如果 s 的第二个标量是 7`，第四个标量是 49，也会通过检查，第一次检查仅仅是为了验证第一个门的输入和输出的一致性。

QAP

Arithmetic circuits can be expressed in a simplified form known as Rank-1 Constraint System (R1CS).

However, R1CS is **not the most efficient way** to verify the correctness of computations expressed as arithmetic circuits.

Therefore, we can **transform R1CS into a Quadratic Arithmetic Program (QAP)**,

$$R1CS \longrightarrow QAP$$

which is a more efficient way to verify the correctness of computations expressed as arithmetic circuits. This transformation from R1CS to QAP allows for more efficient and streamlined methods of checking the correctness of computations expressed as arithmetic circuits.

1. R1CS to QAP

> 代数中间表示
> BTW , 你可以将任何问题表示为 QAP 形式

从 3 个向 4 × 6 的向量转变到 6 组长度为 3 的多项式，

From - A、B、C 的 R1CS :

```
A
[0, 1, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0]
[0, 1, 0, 0, 1, 0]
[5, 0, 0, 0, 0, 1]

B
[0, 1, 0, 0, 0, 0]
[0, 1, 0, 0, 0, 0]
[1, 0, 0, 0, 0, 0]
[1, 0, 0, 0, 0, 0]

C
[0, 0, 0, 1, 0, 0]
```

```
[0, 0, 0, 0, 1, 0]
[0, 0, 0, 0, 0, 1]
[0, 0, 1, 0, 0, 0]
```

To - `polynomials coefficient` :
下面会讲到是如何插值得到这些多项式系数的

```
A polynomials
[-5.0, 9.166, -5.0, 0.833]   ->  means -5 + 9.166x - 5x^2 + 0.833x^3
[8.0, -11.333, 5.0, -0.666]
[0.0, 0.0, 0.0, 0.0]
[-6.0, 9.5, -4.0, 0.5]
[4.0, -7.0, 3.5, -0.5]
[-1.0, 1.833, -1.0, 0.166]
B polynomials
[3.0, -5.166, 2.5, -0.333]
[-2.0, 5.166, -2.5, 0.333]
[0.0, 0.0, 0.0, 0.0]
[0.0, 0.0, 0.0, 0.0]
[0.0, 0.0, 0.0, 0.0]
[0.0, 0.0, 0.0, 0.0]
C polynomials
[0.0, 0.0, 0.0, 0.0]
[0.0, 0.0, 0.0, 0.0]
[-1.0, 1.833, -1.0, 0.166]
[4.0, -4.333, 1.5, -0.166]
[-6.0, 9.5, -4.0, 0.5]
[4.0, -7.0, 3.5, -0.5]
```

系数是升序排序的，例如上述第一个多项式是 : $-5 + 9.166 \times x - 5 \times x^2 + 0.833 \times x^3$, 这个多项式是经过这些点 `(1, 0) (2, 0) (3, 0) (4, 5)` 的拉格朗日插值 得到的 :

```
  A
1 [0, 1, 0; 0, 0, 0]
2 [0, 0, 0, 1, 0, 0]
3 [0, 1, 0, 0, 1, 0]
4 [5, 0, 0, 0, 0, 1]
```

(可以在 [这个网址](https://zh.planetcalc.com/8692/) 代入这些点进行插值计算 :

插值过程 :

For all the first positions of vector A we get (1,0),(2,0),(3,0),(4,5)
By using lagrange interpolation on the above system.

$$p(x_{A1}) = \frac{(x-2)(x-3)(x-4)}{(1-2)(1-3)(1-4)} \times 0 + \frac{(x-1)(x-3)(x-4)}{(2-1)(2-3)(2-4)} \times 0 + \frac{(x-1)(x-2)(x-4)}{(3-1)(3-2)(3-4)} \times 0 + \frac{(x-1)(x-2)(x-3)}{(4-1)(4-2)(4-3)} \times 5$$

$$p(x_{A1}) = 0 + 0 + 0 + \frac{(x^3-6x^2+11x-6)}{6} \times 5$$

$$p(x_{A1}) = 0.833x^3 - 5x^2 + 9.166x - 5$$

我们尝试在 `x=1` 处 (`A` 的第 `1` 行) 评估所有这些多项式。

```
A results at x=1
0   <-- 多项式是: -5 + 9.166x - 5x^2 + 0.833x^3, 代入 1 得 0  ∵ (1,0) 插值
1   <-- 多项式是: 8 + -11.333x + 5x^2 -0.666x^3, 代入 1 得 1  ∵ (1,1) 插值
0
0
0
0
B results at x=1
0
1
0
0
0
0
C results at x=1
0
0
```

```
0
1
0
0
```

正好对应一开始逻辑门构造的 $\mathcal{L}$、$\mathcal{R}$、$\mathcal{O}$ (即 a、b、c) (废话, 就是用他们的值构造的) :

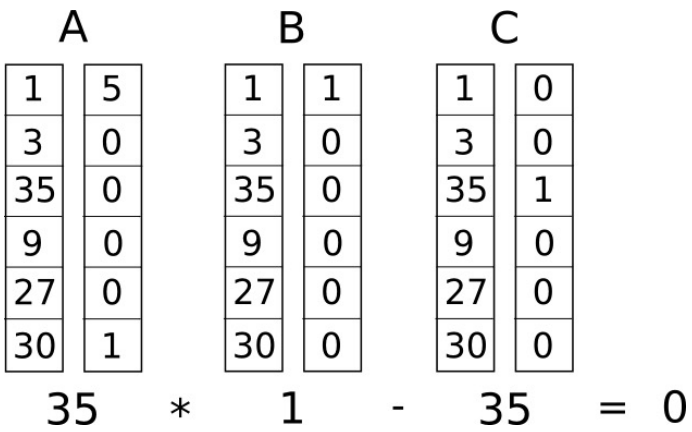
```
  A
1 [0, 1, 0; 0, 0, 0]
2 [0, 0, 0, 1, 0, 0]
3 [0, 1, 0, 0, 1, 0]
4 [5, 0, 0, 0, 0, 1]
```

2. Checking the QAP

Now what's the point of this crazy transformation? The answer is that instead of checking the constraints in the R1CS individually, we can now check *all of the constraints at the same time* by doing the dot product check *on the polynomials*. (我们现在可以通过对多项式进行**点积**检查来同时检查所有约束，而不是单独检查 R1CS 中的约束)

~one (w_0)	x	sym_1	y	sym_2	~out
1	3	9	27	30	35

如图 , `[1, 3, 9, 27, 30, 35 ]` 是 `[~one, x, sym_1, y, sym_2, ~out]` 一系列中间变量的取值, 通常被称为 `witness` (The witness is simply the **assignment** to **all the variables**, including input, output and internal variables)



If the resulting polynomial, evaluated at every x coordinate that we used above to represent a logic gate, is equal to zero, then that means that all of the checks pass; (如果在我们上面用来表示逻辑门的每个 x 坐标处计算的结果多项式等于零，则意味着所有检查都通过；)

> `resulting polynomial` , 应该是指的 `Lagrange interpolation` 生成的多项式 ;

if the resulting polynomial evaluated at at least one of the x coordinate representing a logic gate gives a nonzero value, then that means that the values going into and out of that logic gate are inconsistent (如果在对表示逻辑门的 x 坐标的评估中, 一旦有 1 处评估的 resulting polynomial 给出非 0 值，则这意味着进出该逻辑门的值不一致)

(ie. the gate is `y = x * sym_1` but the provided values might be `x = 2` , `sym_1 = 2` and `y = 5`).

Note that the `resulting polynomial` does not itself have to be zero(生成的多项式本身不必为零), and in fact in most cases won't be;

it could have any behavior at the points that don't correspond to any logic gates, as long as the result is zero at all the points that *do* correspond to some gate. (它可以在不对应于任何逻辑门的点处有任何行为，只要结果在所有对应于某个门的点处为零即可)
我理解就是, 在其他和逻辑门无关的 Point , 多项式想长成什么样都无所谓 ; 但是在逻辑门评估的那几个点, 多项式需要等于 = 0 ; 这简直是废话, 本来多项式就是用那几个点插值插出来的

To check correctness, we don't actually evaluate the polynomial `t = A . s * B . s - C . s` at every point corresponding to a gate; instead, we divide `t` by another polynomial, `Z` , and check that `Z` evenly divides `t` - that is, the division `t / Z` leaves no remainder.
(就是活, 不再去评估整个 R1CS : `t = A . s * B . s - C . s` , 而是利用 t 多项式除 Z , 去看有没有余数)

Z is defined as `(x - 1) * (x - 2) * (x - 3) ...` - the simplest polynomial that is equal to zero at **all points that correspond to logic gates**.

It is an elementary fact of algebra that *any* polynomial that is equal to zero at all of these points has to be a multiple of this minimal polynomial, and if a polynomial is a multiple of Z then its evaluation at any of those points will be zero; this equivalence makes our job much easier. (代数的一个基本事实是，任何经过 (P1, P2 P3) 点多项式必须是经过这些点的最小多项式的**倍数**，比如 $x^3 - 1 = 0$ 经过 $(1, 0)$, 则 $x^3 - 1$ 一定是 $x - 1$ 的倍数. 如果一个多项式是 Z 的倍数，那么它在任何Z 的这些点的评估都将为零; 这种等价性使我们的工作容易得多。)

3. dot product check

现在，让我们用上面的多项式实际做点积检查。首先，中间多项式：

```
A . s = [43.0, -73.333, 38.5, -5.166]
B . s = [-3.0, 10.333, -5.0, 0.666]
C . s = [-41.0, 71.666, -24.5, 2.833]
```

这是怎么来的呢 ???

recap 下 `A polynomials` 和 `witness` ：

```
A polynomials
[-5.0, 9.166, -5.0, 0.833]
[8.0, -11.333, 5.0, -0.666]
[0.0, 0.0, 0.0, 0.0]
[-6.0, 9.5, -4.0, 0.5]
[4.0, -7.0, 3.5, -0.5]
[-1.0, 1.833, -1.0, 0.166]

witness:
[1, 3, 9, 27, 30, 35 ]
```

运算链接
详细运算过程

$$\begin{pmatrix} -5 & 9+\frac{1}{6} & -5 & \frac{5}{6} \\ 8 & -11-\frac{1}{3} & 5 & -\frac{2}{3} \\ 0 & 0 & 0 & 0 \\ -6 & 9.5 & -4 & 0.5 \\ 4 & -7 & 3.5 & -0.5 \\ -1 & \frac{11}{6} & -1 & \frac{1}{6} \end{pmatrix}^T \begin{pmatrix} 1 & 3 & 35 & 9 & 27 & 30 \end{pmatrix}^T$$

$$= \begin{pmatrix} -5 & 8 & 0 & -6 & 4 & -1 \\ 9+\frac{1}{6} & -11-\frac{1}{3} & 0 & 9.5 & -7 & \frac{11}{6} \\ -5 & 5 & 0 & -4 & 3.5 & -1 \\ \frac{5}{6} & -\frac{2}{3} & 0 & 0.5 & -0.5 & \frac{1}{6} \end{pmatrix} \begin{pmatrix} 1 \\ 3 \\ 35 \\ 9 \\ 27 \\ 30 \end{pmatrix} = \begin{pmatrix} 43 \\ -73.33333\ldots \\ 38.5 \\ -5.16666\ldots \end{pmatrix}$$

A 尚如此, B/C 同理 ;

`A . s * B . s - C . s` : 得 :

```
t = [-88.0, 592.666, -1063.777, 805.833, -294.777, 51.5, -3.444]
```

这是怎么算出来的呢 ?

`A·s=[43.0, -73.333, 38.5, -5.166]` `B·s = [-3.0, 10.333, -5.0, 0.666]` `C·s = [-41.0, 71.666, -24.5, 2.833]`

A·s　　B·s　　C·s

$$(-(5+\frac{1}{6})x^3+38.5x^2-(73+\frac{1}{3})x+43)(\frac{2}{3}x^3-5x^2+(10+\frac{1}{3})x-3)-((2+\frac{5}{6})x^3-24.5x^2+(71+\frac{2}{3})x-41)$$
$$=-3.44444\ldots x^6+51.5x^5-294.77777\ldots x^4+805.83333\ldots x^3-1063.77777\ldots x^2+592.66666\ldots x-88$$

```
t = [-88.0,  592.666,  -1063.777,  805.833,  -294.777,  51.5,  -3.444]
```

Now, the minimal polynomial `Z = (x - 1) * (x - 2) * (x - 3) * (x - 4)` :

$$(x-1)\cdot(x-2)\cdot(x-3)\cdot(x-4)$$
$$=x^4-10x^3+35x^2-50x+24$$

取上式系数(倒一下顺序) : `Z = [24, -50, 35, -10, 1]`

得: `h = t / Z = [-3.666, 17.055, -3.444]` , 这特么又是怎么来的呢 ???

- t : `[-88.0, 592.666, -1063.777, 805.833, -294.777, 51.5, -3.444]`
- Z : `[24, -50, 35, -10, 1]`

t 是多项式的系数 ;
Z 是 $(x-1)\cdot(x-2)\cdot(x-3)\cdot(x-4)$ 多项式的系数 ;
h 是多项式除法得到的低次多项式 (为什么可以整除: 因为 Z 插值的这几个点都是 t 上评估为 0 的点)

$$
\begin{aligned}
& h * Z = t \\
& = (-3.444x^2 + 17.055x - 3.666) * (x-1)\cdot(x-2)\cdot(x-3)\cdot(x-4) \\
& = -3.444x^6 + 51.5x^5 - 294.7x^4 + 805x^3 - 1063.7x^2 + 592.6\ldots x - 87.984
\end{aligned}
$$

4. Summary

And so we have the solution for the QAP.

If we try to falsify(伪造) any of the variables in the R1CS solution that we are deriving this QAP solution from — say, set the last one to `31` instead of `30`, then we get a `t` polynomial that fails one of the checks (x=3 处的结果将等于 `-1` 而不是 `0`),

而且 , t 不会是 Z 的倍数；相反， `t / Z` 无法整除 , 将得到 `[-5.0, 8.833, -4.5, 0.666]` 的余数。

Note that the above is a simplification;
the addition, multiplication, subtraction and division will happen not with regular numbers, but rather with `finite field elements`

so all the algebraic laws we know and love still hold true, but where all answers are elements of some finite-sized set, usually integers within the range from 0 to n-1 for some n.

For example, if n = 13, then 1 / 2 = 7 (and 7 *2 = 1), 3* 5 = 2, and so forth.

Using finite field arithmetic removes the need to worry about rounding errors and allows the system to work nicely with `elliptic curves`, which end up being necessary for the rest of the zk-SNARK machinery that makes the zk-SNARK protocol actually secure.

AIR

详见 STARK-2

step	a	b
i=1	1	1
i=2	2	3
i=3	5	8
i=4	13	21

转化程序：

$$
\begin{aligned}
f_1(X_1, X_2, {X_1}^{next}, {X_2}^{next}) &= A^{next} - (B + A) \\
f_2(X_1, X_2, {X_1}^{next}, {X_2}^{next}) &= B^{next} - (B + A^{next})
\end{aligned}
$$

举例：如上图，第 $i = 2$ 行的转换：

$$
\begin{aligned}
f_1(X_1, X_2, {X_1}^{next}, {X_2}^{next}) &= 5 - (3 + 2) = 0 \\
f_2(X_1, X_2, {X_1}^{next}, {X_2}^{next}) &= 8 - (3 + 5) = 0
\end{aligned}
$$

> 其中 A, B 是对 a, b 列元素进行拉格朗日插值得到的多项式函数。

现在，要将该 Fibonacci 程序修改至宽度为 3 的 AIR：

step	a	b	c
i=1	1	1	2
i=2	3	5	8
i=3	13	21	34
i=4	55	89	144

转化程序：

$$
\begin{aligned}
f_1(X_1, X_2, X_3, {X_1}^{next}, {X_2}^{next}, {X_3}^{next}) &= A^{next} - (B + C) \\
f_2(X_1, X_2, X_3, {X_1}^{next}, {X_2}^{next}, {X_3}^{next}) &= B^{next} - (C + A^{next}) \\
f_3(X_1, X_2, X_3, {X_1}^{next}, {X_2}^{next}, {X_3}^{next}) &= C^{next} - (A^{next} + B^{next})
\end{aligned}
$$

第 $i = 2$ 行举例：

$$A^{next} - (B + C) = 13 - (5 + 8) = 0$$
$$B^{next} - (C + A^{next}) = 21 - (8 + 13) = 0$$
$$C^{next} - (A^{next} + B^{next}) = 34 - (13 + 21) = 0$$

Preprocessed AIR (PAIR)

Randomised AIR with Preprocessing (RAP)

RAP 是为了更有效地执行计算或协议，同时保持零知识性质而开发的。

RAP 是 AIR 的一种扩展，它增加了随机化和预处理步骤。

- 随机化（Randomisation）是为了增加证明的安全性。在生成证明时，会引入一些随机数，这使得攻击者即使知道证明的构造方式，也无法确定具体的解。这对于保持证明的零知识性质非常重要。
- 预处理（Preprocessing）是为了提高证明的效率。在预处理步骤中，会预先计算一些可以重复使用的数据，这样在生成证明时就可以直接使用这些数据，而不需要再次计算。这可以大大提高证明的生成和验证速度。

Randomized AIR with Preprocessing：

step	a	b	c
$i=1$	a_1	b_1	1
$i=2$	a_2	b_2	$\frac{a_1+\gamma}{a_2+\gamma}$
$i=3$	a_3	b_3	$\frac{(a_1+\gamma)(a_2+\gamma)}{(a_2+\gamma)(a_3+\gamma)}$
$i=4$	0	0	$\frac{(a_1+\gamma)(a_2+\gamma)(a_3+\gamma)}{(a_2+\gamma)(a_3+\gamma)(a_1+\gamma)}$

目标：多重集合相等性检查

1. 列 a 和 b 包含了你想要检查相等性的两个多重集合的元素，而列 c 是用来构建你的零知识证明的。
2. 随机数 γ 用于随机化你的证明 ，增加其安全性
3. 使用了一个特定的函数（在这个例子中是乘积函数）来将列 a 和 b 转化为列 c . 这个函数是你的约束多项式，它定义了 a 和 b 之间的关系。
4. 在每一行 i中，都计算了 $\prod_{1\leq j\leq i}(a_j+\gamma)/(b_j+\gamma)$ 并将结果存储在 c_i —— 生成新列 z 。
5. 最后，检查 $Z^{next}\cdot(B+\gamma)-Z\cdot(A+\gamma)=0$ ，确保 a 和 b 是相等的。如果这个约束不满足，那么你就知道 a 和 b 不相等。如果满足，那么你就得到了一个证明 a 和 b 相等的零知识证明。

这个过程是一个标准的 RAP 过程，它使用了随机化（ γ ）和预处理（通过计算列 c）来生成一个零知识证明。这个证明可以被任何人验证，但是它不揭露任何关于 a 和 b 的具体信息，只揭露了它们是否相等。

约束多项式为：

$$\prod_{i\in[n]}(a_i+\gamma)=\prod_{i\in[n]}(b_i+\gamma)\Longrightarrow\prod_{i\in[n]}(a_i+\gamma)/(b_i+\gamma)=1$$

构建 z 列：

$$z_i=\prod_{1\leq j\leq i}(a_j+\gamma)/(b_j+\gamma)$$

检查约束：

$$Z^{next}\cdot(B+\gamma)-Z\cdot(A+\gamma)=0$$

假如第 $i=2$ 行：

$$\begin{aligned}&\frac{(a_1+\gamma)(a_2+\gamma)}{(a_2+\gamma)(a_3+\gamma)}\cdot(a_3+\gamma)-\frac{(a_1+\gamma)}{(a_2+\gamma)}\cdot(a_2+\gamma)\\&=(a_1+\gamma)-(a_1+\gamma)\qquad//\text{约分}\\&=0\end{aligned}$$

仅在行 $i=1$ 上应用RAP的多重集合相等性检查的约束：

生成两列元素 R 和 S ，其中 S 只有 $S_1=1$ ，其他 S 等于0， R 只有 $R_1=0$ ，其他 R 均等于1。

令约束多项式函数为：

$$\prod_{i\in[n]}(S_i\cdot(a_i+r)/(b_i+r)+R_i)=1$$

- 当 $i=1$ 该约束退化为标准的多重集合相等性检查的约束性函数；
- 当 $i>1$ 时，该约束两边恒相等，综上该约束只在 $i=1$ 时生效。

在这个函数中，a_i 和 b_i 是我们要检查相等性的两个集合的元素，r 是一个随机数。

- 当 $i=1$ 的时候，由于 $S_1=1$ 和 $R_1=0$ ，这个函数就变成了 $\frac{a_1+r}{b_1+r}=1$ ，这是一个标准的集合相等性检查的约束。
- 当 $i>1$ 的时候，由于 $S_i=0$ 和 $R_i=1$ ，这个函数就变成了 $1=1$，这是一个恒等式，所以不会影响结果。

这样，我们就得到了一个只在 $i = 1$ 上生效的约束。这个约束能够在不揭示集合内容的情况下，检查 a_1 和 b_1 是否相等。这是一种在零知识证明中常用的技巧，它允许我们在保证隐私的同时，验证一些重要的性质。

Reference :

- https://medium.com/starkware/arithmetization-i-15c046390862
- https://medium.com/@VitalikButerin/quadratic-arithmetic-programs-from-zero-to-hero-f6d558cea649
- https://www.jianshu.com/p/c81cb6c01d76
- https://tlu.tarilabs.com/cryptography/rank-1#arithmetic-circuits
- http://learn.0xparc.org/materials/circom/additional-learning-resources/r1cs%20explainer/