Circom (zkpl)

Circom syntax

Circom 的语法类似 javascript 和 C ，提供一些基本的数据类型和操作，例如 `for、while、>>、array` 等。

关键字：

- `signal`：信号变量，要转换为电路的变量，可以是 `private` 或 `public`
- `template`：模板，用于函数定义，就像 JS / Solidity 中的 `function`
- `component`：组件变量，可以把组件变量想象成对象，而信号变量是对象的公共成员

Circom 也提供了一些操作符用于操作信号变量：

- `<--, -->`：这些操作符为信号变量赋值，但不会生成约束条件
- `===` ：这个操作符用来定义约束
- `<==, ==>`：这两个操作符用于连接信号变量，同时定义约束

1. `<==` means `<--` + `===`，即 `赋值` + `定义约束`
2. `==>` 是一种连接运算符，用于实现信号的传递和组件的互连。它表示将左侧的 `output` 连接到右侧信号端口的 `input` 上：

```rust
// 将输入信号 `in` 连接到组件 `myComponent` 的输入上，
// 并将组件 `myComponent` 的输出连接到输出信号 `out` 上。
in ==> myComponent.in;
myComponent.out ==> out;
```

oxPARC github

https://github.com/0xPARC/circom-ecdsa/tree/master/circuits

```
pragma circom 2.1.2;

include "circomlib/poseidon.circom";
// include "https://github.com/0xPARC/circom-secp256k1/blob/master/circuits/bigint.circom";

template Example () {
    signal input in;
    signal input b0;
    signal input b1;
    signal input b2;
    signal input b3;
    in === 8*b3 + 4*b2 + 2*b1 + b0;

    // 如下式子想要保证 bi 为 0/1，不能是其他数字
    0 === b0 * (b0 - 1);
    0 === b1 * (b1 - 1);
    0 === b2 * (b2 - 1);
    0 === b3 * (b3 - 1);
}

component main { public [ b0, b1, b2, b3 ] } = Example();

/* INPUT = {
    "in": "11",
    "b0": "1",
    "b1": "1",
    "b2": "0",
    "b3": "1"
} */
```

改进 - 使用 array 和 for loop

```
template Num2Bits (nBits) {
    signal input in;
    signal input b[nBits];

    var accum = 0;
    for(var i = 0; i < nBits; i++){
        accum += (2 ** i) * b[i];
    }
    in === accum;

    for(var i = 0; i < nBits; i++){
        0 === b[i] * (b[i] - 1);
    }
}

component main { public [ b ] } = Num2Bits(5);

/* INPUT = {
    "in": "11",
    "b": ["1", "1", "0", "1", "0"]
}  */
```

Constraint & assignment :

```
template Example () {
    signal input x1;
    signal input x2;
    signal input x3;
    signal input x4;
    signal y1;
    signal y2;

    signal output out;
    y1 <== x1 + x2;
    y2 <== y1 * x3;
    out <== y2 - x4;     // (2+4)*8-5 == 43
}

component main = Example();

/* INPUT = {
    "x1": "2",
    "x2": "4",
    "x3": "8",
    "x4": "5",
}  */
```

输出 :

```
OUTPUT:
  `out = 43`
```

Last feature :

- 功能 :
  - `Num2Bits` 将数字 Num (如 `11`) 转化为 Bits 二进制形式 如 `01011` , 对应 `b[]` 为 `[1,1,0,1,0]` (倒过来的) :
    - $(11/2^0)\ \%\ 2 = 1; \quad (11/2^1)\ \%\ 2 = 5\%2 = 1; \quad (11/2^2)\ \%\ 2 = 2\%2 = 0.\dots$
  - `Main` template 则将 input signal `in` 传递给 `Num2Bits` 进行二进制转换，并将 `Num2Bits` 的输出信号 `b[]` 的 `b[1]` 和常数 `3` 相加作为输出信号 `out` 。
- 语法 :
  - `b[i] <== (in \ 2 ** i) % 2` :

        - “`in`” 通常是一个包含二进制数的变量。
        - “ `\2 ** i` ”将二进制数除以 `2` 的 `i` 次方。 这相当于将二进制数右移 `i` 位。
        - “`%2`” 即取除 `2` 的余数，相当于只保留二进制数的最后一位。
    - `accum` 的作用：`accum` 变量将二进制数组 `b[]` 转换回对应的整数，而后将输入信号 in 与 accum 进行等值约束以保证过程的正确性。
    - `0 === b[i] * (b[i] - 1)`：
        - 这一步约束是为了确保二进制数组的每一位只能是 0 或 1
- 输出：
    - 输入 `in` 为 `11`，其 5 位二进制表示为 `01011`
    - `b[1] === 1`，`b[1]+3 === 4`，所以输出是 4；

```
pragma circom 2.1.2;

include "circomlib/poseidon.circom";
// include "https://github.com/0xPARC/circom-secp256k1/blob/master/circuits/bigint.circom";

template Num2Bits (nBits) {
    signal input in;
    signal output b[nBits];

    for(var i = 0; i < nBits; i++) {
        // <== : error Non quadratic constraints are not allowed!
         b[i] <== (in \ 2 ** i) % 2;
    }

    var accum = 0;
    for(var i = 0; i < nBits; i++){
        accum += (2 ** i) * b[i];
    }
    in === accum;

    for(var i = 0; i < nBits; i++){
        0 === b[i] * (b[i] - 1);
    }
}

template Main() {
    signal input in;
    signal output out;
    component bitifier = Num2Bits(5);
    bitifier.in <== in;
    out <== bitifier.b[1] + 3;
}

component main = Main();

/* INPUT = {
    "in": "11"
}  */
```

`circom` 允许程序员定义算术电路的约束。

所有约束必须采用 `A * B + C = 0` 的形式，其中 A、B 和 C 是信号的线性组合。

定义我们的第一个电路，它简单地将两个输入信号相乘并产生一个输出信号：

```
pragma circom 2.0.0;
/*This circuit template checks that c is the multiplication of a and b.*/
template Multiplier2 () {
    // Declaration of signals.
```

```
    signal input a;
    signal input b;
    signal output c; // Constraints.
    c <== a * b;
}
```

- 首先， `pragma` 指令用于指定编译器版本（类似于 solidity）。这是为了确保电路与 `pragma` 指令后的编译器版本兼容。否则，编译器会抛出警告。
- 关键字 `template` 来定义新电路的形状，为 `Multiplier2`。
- 定义信号 `signal` 。`signal` 可用 `a, b, c` 等标识符命名
- 这个电路有 2 个 `private` 输入信号，名为 `a` 和 `b` ，还有一个输出信号 `c`。 输入和输出使用 ⇐ 运算符进行关联。
- 在 circom 中， `<==` 运算符做两件事。 一是连接信号。 二是施加约束。
- 在本例中，我们使用 `<==` 将 `c` 连接到 `a` 和 `b` ，同时将 `c` 约束为 `a * b` 的值，即电路做的事情是让强制信号 `c` 为 `a*b` 的值。

编译电路

上面定义了叫 `Multiplier2` 的 `template` 电路。

要实际创建电路，我们必须创建此 `template` 的一个实例（使用名为 `main` 的组件实例化它）：

```
component main = Multiplier2();
```

... 一些命令行操作：构建你的第一个零知识 snark 电路（Circom2） - 登链社区的文章 - 知乎
https://zhuanlan.zhihu.com/p/556765252

算术电路：零知识证明核心

零知识程序和其他程序的实现不太一样。首先，你要解决的问题需要先转化成多项式，再进一步转化成电路。例如，多项式x^3 + x +5 可以表示成如下的电路：

```
sym_1 = x * x // sym_1 = x²
sym_2 = sym_1 * x // sym_2 = x³
y = sym_2 + x // y = x³ + x
~out = y + 5
```

Circom 编译器将逻辑转换为电路。通常我们不需要自己设计基础电路。
如果你需要一个哈希函数或签名函数，可以在 circomlib(一个 JS 库) 找到。

Generate/Verify Proofs

We need to create a trusted `setup` before running zkp programs.

> 在 Run zk 程序之前, 需要做一些可信 `setup` 的创建

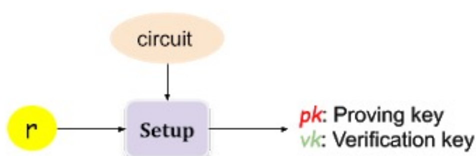
The creation of a trusted setup requires circuits and some random numbers.

> 可信 setup 的创建需要电路以及一些随机数

Once the setup is finished, a **proving key**(证明密钥) and a **verification key**(验证密钥) will be generated.

As the name implies, one is required for generating proofs, and the other is required for verification. (These two keys are different from public/private keys in ECC).

> 顾名思义，一个是用于生成证明，另一个用于验证。(这两个密钥与 ECC(Elliptic-curve cryptography 椭圆曲线密码学) 中的公钥/私钥不同）。



一旦创建了 **proving key**(证明密钥) 和 **verification key**(验证密钥)，就可以生成 **proofs** (证明) 了！

There are two kinds of inputs, `public` and `private`：

- 比如 A 转账给 B 但不想泄露余额，那么 A 的余额就是 `private input`，**也叫** `witness` 。
- `public input` 可以是 A 和 B 的 address 或转账 amount， 这取决于算法设计。

然后， `Provers` 通过 **proving key**、**public input** 和 **witness** 生成证明。

最后一步即 `verification` 验证

Verifiers verify the proof by **public inputs**, the **proof** and the **verification key**.

> Verifiers 通过 **public inputs**, the **proof** and the **verification key** 来验证 Proof

Public input, witness(private input), proving key, verification key, circuits(电路), proof and their **relations** are the thing you need to know!

Exercise

Num2Bits

- Parameters: `nBits`
- Input signal(s): `in`
- Output signal(s): `b[nBits]`

The output signals should be an array of bits of length `nBits` equivalent to the binary representation of `in`. `b[0]` is the least significant bit. (`b[0]` 是最低有效位)

Solution :

```
template Num2Bits(n) {
    signal input in;
    signal output out[n];
    var lc1=0;  // 累加器, make sure 最终的二进制转换结果的整数值与 n 相等

    var e2=1;
    for (var i = 0; i<n; i++) {
        out[i] <-- (in >> i) & 1; // 将 `in` 右移 `i` 位, 然后与 1 进行按位与操作
        out[i] * (out[i] -1 ) === 0; // 约束值为 0 or 1
        lc1 += out[i] * e2;  // 累加器累加结果
        e2 = e2+e2;  // e2 * 2: 其值: 1,2,4,8,16 ....
    }

    lc1 === in;
}
```

```
template Num2Bits_strict() {
    signal input in;
    signal output out[254];

    component aliasCheck = AliasCheck();
    component n2b = Num2Bits(254);
    in ==> n2b.in;
```

```
    for (var i=0; i<254; i++) {
        n2b.out[i] ==> out[i];
        n2b.out[i] ==> aliasCheck.in[i];
    }
}
```

IsZero

- Parameters: none
- Input signal(s): `in`
- Output signal(s): `out`

Specification: If `in` is zero, `out` should be `1`. If `in` is nonzero, `out` should be `0`. This one is a little tricky!

下面代码为什么这么复杂? 因为使用了信号处理的常见技巧，即: 使用一个 `input signal` 的倒数来检查其是否为零。

> 由于信号中可能包含噪声和误差，因此我们不能直接比较信号是否等于零。相反，一种更可靠的方法是检查输入信号的倒数是否趋近于无穷大。

然而，在电路中，不能直接使用除法 (我理解是, 在 `<==` 和 `==>` 的两侧不能出现除法 , 但是 `-->` 和 `<--` 可以),

通过将 `in` 处理为其倒数，可以避免在电路中出现除以 0 的错误。

最后， `in*out === 0` 的意思是输入信号和输出信号必须是相反数或其中一个为零。这个条件是为了确保输出信号是输入信号的相反数或为零。

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

    signal inv;

    inv <-- in!=0 ? 1/in : 0;

    out <== -in*inv +1;
    in*out === 0;
}
```

https://github.com/Antalpha-Labs/zkp-co-learn/discussions/21 :

1. 只有 `<==` / `===` / `==>` 才会被编译到最后的约束系统里
2. `<--` 和一些其他的计算，是为了方便根据 input 生成 prove用的，即使没有这些计算，prover 手动算出来放到最后 proving 过程里也可以。
3. 之前的错误答案里的计算和赋值，只是实现了“prover不作恶情况下如何根据input计算出正确的值”，并不能防止“恶意的prover用错误值通过verify”， 因为这些没有进入最后的约束里。

inv 是个辅助输入
var 是用来辅助收集 linear combanation

5的inverse 是不是1/5呀？为什么是这么大的数字呢？
8755297148735710088898562298102910035419345760166413737479281674630323

> 因为是有限域里的-1次方

IsEqual

- Input signal(s): `in[2]`
- Output signal(s): `out`

Specification: If `in[0]` is equal to `in[1]`, `out` should be `1`. Otherwise, `out` should be `0`.

```
template IsEqual() {
    signal input in[2];
    signal output out;
```

```
    component isz = IsZero();

    in[1] - in[0] ==> isz.in;

    isz.out ==> out;
}
```

LessThan

- Input signal(s): `in[2]`. Assume that it is known ahead of time that these are at most 2252−12252−1.
- Output signal(s): `out`
  Specification: If `in[0]` is strictly less than `in[1]`, `out` should be `1`. Otherwise, `out` should be `0`.

  > 如果 `in[0]` 严格小于 `in[1]` ，out 应该是 1。否则，out 应该是 0 。

```
template LessThan(n) {
    assert(n <= 252); // 需要 n <= 252
    signal input in[2];
    signal output out;

    component n2b = Num2Bits(n+1);

    n2b.in <== in[0]+ (1<<n) - in[1];

    out <== 1-n2b.out[n];
}
```

`circom` install

核心工具是用 Rust 编写的 `circom` 编译器。

tutorial for install circom/snarkjs

```
// which variable are PUBLIC to the Verifier when a proof is generated
// : it's `out`
component main { public[out] } = Example();
```

ZK Rebel allows you to test your circuit very rapidly on inputs in comment JSON format :

```
/* INPUT = {
    "a": "5",
    "b": "77"
} */
```

```
pragma circom 2.1.2;

include "circomlib/poseidon.circom";
// include "https://github.com/0xPARC/circom-secp256k1/blob/master/circuits/bigint.circom";

template Example () {
    signal input a;
    signal input b;
    signal output c;

    var unused = 4;
    c <== a * b;
    assert(a > 2);

    component hash = Poseidon(2);
```

```
    hash.inputs[0] <== a;
    hash.inputs[1] <== b;

    log("hash", hash.out);
}

component main { public [ a ] } = Example();

/* INPUT = {
    "a": "5",
    "b": "77"
} */
```

待整理: circom 最终编译成 ⇒ 多项式约束：
里面只有变量和一些参数，变量的部分都必须只有+和*，参数部分就是你可以随意各种计算的。
你把底层搞懂了再往上看circom，其实可以看成一个辅助翻译器

Ref :

- https://medium.com/coinmonks/hands-on-your-first-zk-application-70fe3a0c0d82
- zk-snark新手入门教程【circom/snarkjs】 - 汇智网的文章 - 知乎 https://zhuanlan.zhihu.com/p/143519030
- 构建你的第一个零知识 snark 电路（Circom2） - 登链社区的文章 - 知乎 https://zhuanlan.zhihu.com/p/556765252