R1CS Programming

ZK0x04 Workshop Notes

Daniel Lubarov Brendan Farmer

October 19, 2019

Contents

1 Multiplicative inverse

Deterministically computing $1/x$ in an R1CS circuit would be expensive. Instead, we can have the prover compute $1/x$ outside of the circuit and supply the result as a witness element, which we will call x_{inv}. To verify the result, we enforce

$$(x)(x_{\text{inv}}) = (1) \tag{1}$$

2 Zero testing

To assert $x = 0$, we simply enforce

$$(x)(1) = (0) \tag{2}$$

Asserting $x \neq 0$ is similarly easy: we compute $1/x$ (non-deterministically, as in Section 1). The result can be ignored; the mere fact that an inverse exists implies $x \neq 0$.

On the other hand, if we want to *evaluate*

$$y \coloneqq \begin{cases} 0 & \text{if } x = 0, \\ 1 & \text{otherwise,} \end{cases} \tag{3}$$

we can do so by introducing another variable, m, and enforcing

$$(x)(m) = (y), \tag{4}$$
$$(1 - y)(x) = (0). \tag{5}$$

Outside of the circuit, the prover generates y as in Equation 3, and generates m as

$$m \coloneqq \begin{cases} 1 & \text{if } x = 0, \\ y/x & \text{otherwise.} \end{cases} \tag{6}$$

This method is from [1].

3 Binary

To assert $b \in \{0, 1\}$, we enforce

$$(b)(b - 1) = (0). \tag{7}$$

To convert a field element x to its binary encoding, $(b_1, \ldots, b_n)$, we have the prover generate the binary encoding out-of-band. We then verify it by applying Equation 7 to each b_i, and enforcing

$$(x)(1) = \left(\sum_{i=0}^{n-1} 2^i b_i \right), \tag{8}$$

assuming a little-ending ordering of the bits.

Note that Equation 8 permits two encodings of certain field elements. In $\mathbb{F}_{13}$ for example, the element 1 can be represented as either 0001 or 1110. If a canonical encoding is required, we can prevent "overflowing" encodings by asserting that $(b_1, \ldots, b_n) < |F|$. Such binary comparisons are covered in Section 9.

4 Selection

Suppose we have a boolean value s, and we wish to compute

$$z \coloneqq \begin{cases} x & \text{if } s = 0, \\ y & \text{if } s = 1. \end{cases} \tag{9}$$

We can compute this as

$$z \coloneqq x + s(y - x). \tag{10}$$

This requires two constraints: one "is boolean" asertion (Equation 7), assuming s was not already known to be boolean, and another for the multiplication.

5 Random access

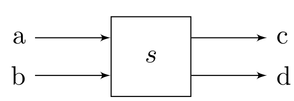
TODO: Discuss naive random access via index comparisons

TODO: Discuss binary tree method

TODO: Discuss single-constraint method for $n = 4$.

6 2x2 switch

Suppose we wish to implement a switch with the following structure:



In particular, if $s = 0$ then the outputs should be identical the inputs: $(c, d) = (a, b)$. If $s = 1$ then the inputs should be swapped: $(c, d) = (b, a)$.

This requires two constraints: one "is boolean" assertion (Equation 7), and another for selecting the value of c (Equation 10). Once we have c, we can compute d as

$$d \coloneqq a + b - c \tag{11}$$

which does not require any additional constraints.

7 Permutations

Say we want to verify that two sequences, $(x_1, \ldots, x_n)$ and $(y_1, \ldots, y_n)$, are permutations of one another. This can be done efficiently using routing networks, which used a fixed (for a fixed n) network of 2x2 switches.

AS-Waksman networks [2] are a particularly useful construction, since they support arbitrary permutation sizes. They use about $n \log_2(n) - n$ switches, which is close to the theoretical lower bound of $\log_2(n!)$.

8 Sorting

Like permutation networks, sorting networks use a fixed network of gates. In particular, a sorting network is comprised of several 2x2 comparator gates, each of which takes two inputs and sorts them. It is theoretically possible to construct a sorting network for n elements using $\mathcal{O}(n \log n)$ gates [3], but practical constructions use $\mathcal{O}(n \log^2 n)$ gates. Since each comparison adds $\mathcal{O}(\log |F|)$ constraints, this approach is fairly expensive.

A better solution is to leverage non-determinism: instead of creating an R1CS circuit to sort a sequence, we have the prover supply the ordered sequence. Using a permutation network (Section 7), we can efficiently verify that the two sequences are permutations of one another. Then for each contiguous pair of elements in the ordered sequence, x_i and x_{i+1}, we assert $x_i \leq x_{i+1}$.

9 Comparisons

TODO: Describe the basic comparison algorithm.

TODO: Describe Ahmed's optimization.

A couple other optimizations are possible in particular circumstances:

1. To assert (not evaluate) $x < y$, we can split x non-canonically and split y canonically. The prover is forced to use x's canonical representation anyway, otherwise $x_{\text{bin}} \geq |F| > y_{\text{bin}}$, making the assertion unsatisfiable.

2. To assert $x < c$ for some constant $c \ll |F|$, we can split x into just $\lceil \log_2 c \rceil$ bits.

10 Embedded curve operations

Embedded curves have several uses in SNARKs. A few examples are Schnorr signatures, Pedersen hashes, and recursive SNARK verifiers. Here we will focus on twisted Edwards curves such as Jubjub.

10.1 Addition

Recall the addition law for twisted Edwards curves,

$$(x_1, y_1) = \left(\frac{x_1 y_1 + y_1 x_2}{1 + d x_1 x_2 y_1 y_2}, \frac{y_1 y_2 - a x_1 x_2}{1 - d x_1 x_2 y_1 y_2} \right). \tag{12}$$

Applying the law directly takes 7 constraints: 4 for the products in the numerators, one for the denominator product, and one[1] for each of the two quotients.

TODO: Discuss constant addition, which takes just 3 constraits.

10.2 Doubling

TODO: Explain the cost of doubling.

10.3 Multiplication

TODO: Discuss multiplication by doubling, along with its variants, like windowed multiplication.

References

[1] B. Parno, J. Howell, C. Gentry, and M. Raykova, "Pinocchio: Nearly practical verifiable computation," in *2013 IEEE Symposium on Security and Privacy*, pp. 238–252, IEEE, 2013.

[2] B. Beauquier and E. Darrot, "On arbitrary size waksman networks and their vulnerability," *Parallel Processing Letters*, vol. 12, no. 03n04, pp. 287–296, 2002.

[3] M. Ajtai, J. Komlós, and E. Szemerédi, "An 0 (n log n) sorting network," in *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pp. 1–9, ACM, 1983.

[1] In general, computing a quotient $q \coloneqq x/y$ takes two constraints: $(y)(y_{\text{inv}}) = (1)$ and $(x)(y_{\text{inv}}) = (q)$. In this case, however, we can multiply both sides by the denominator since we know it will never be zero. This yields a single constraint: $(q)(y) = (x)$.