PLONK with lookups (draft)

Ariel Gabizon
Aztec

Zachary J. Williamson
Aztec

February 29, 2020

1 Introduction

We can use a reduced form of PLONK's permutation argument, to validate that two polynomials share the same roots. This in turn can be used to verify two polynomials share the same Lagrange coefficients.

For two degree-n polynomials $A(X) = \sum_{i=1}^{n} a_i L_i(X), B(X) = \sum_{i=1}^{n} b_i L_i(X)$. Using a random challenge $\gamma \in \mathbb{F}_p$, we can verify that the two sets $\{a_1, \ldots, a_n\}, \{b_1, \ldots, b_n\}$ are identical:

$$\prod_{i=1}^{n} \frac{(a_i + \gamma)}{(b_i + \gamma)} = 1$$

In addition, we can validate the correctness of subsets within larger sets. For l subsets, we introduce a new challenge β and two selector polynomials $q_1(X) = \sum_{i=1}^{n} \sigma_1(i) L_i(X), q_2(X) = \sum_{i=1}^{n} \sigma_2(i) L_i(X)$, where both $\sigma_1(i)$ and $\sigma_2(i)$ evaluate on H to l unique values. The modified permutation argument becomes

$$\prod_{i=1}^{n} \frac{(a_i + \sigma_1(i)\beta + \gamma)}{(b_i + \sigma_2(i)\beta + \gamma)} = 1$$

2 Implementing arbitrary lookup tables

Consider a PLONK circuit with a program width of m and a depth of n. We can represent witnesses to the circuit's memory cells via w_i for all $i \in [1, \ldots, mn]$.

We wish to prove that a subset of memory cells are copies of elements in a size k precomputed lookup table. We represent table values via t_i for all $i \in [1, \ldots, k]$.

To ensure that all polynomial identities are evaluated modulo the vanishing polynomial of a size-n multiplicative subgroup H, we can split the lookup table into size-n

subsets. i.e. our PLONK circuit now has a *table* width of v, where $vn >= k$.

To validate a sequence of lookups, the high level outline is as follows

1. create a key/value mapping that maps an index in the lookup table to a value
2. apply the key/value mapping to the table reads in program memory
3. isolate *unique* table reads in program memory
4. isolate values from the lookup table t that were copied into program memory
5. validate a subset permutation on the set of unique table reads and the set of used table lookup values

3 Using randomized constraints for key/value mappings

We can use randomized constraints to create key/value mappings for arbitrary combinations of keys and values. This is useful for situations where a key and a value require more values than the set of integers $[1, \ldots, p]$, where p is the order of the circuit's native field arithmetic.

After committing to program memory values, the verifier generates a random challenge α and sends it to the prover. We can discriminate between l distinct lookup tables using the l'th powers of α.

The prover can now commit to a sorted list of table reads. We can define the number of table reads as $e = sn$, where $s <= w$. The set of table reads is represented via $\{r_1, \ldots, r_e\}$, where each table read is a linear combination of two memory cells w_1, w_2, where $r = w_1 + \alpha w_2$. The specific mapping between memory cells and table read elements is described using l randomized constraints, represented by identities $E_1, \ldots, E_l$.

We can now validate a permutation between the set of all table reads

$$r_1, \ldots, r_e$$

, and the set of values produced by applying

$$E_1, \ldots, E_l$$

to every row in program memory.

example: XOR gates, width-4 circuit. Consider the scenario where we wish to evaluate $a^b = c$, where a, b, c are in the range $[0, \ldots, 255]$. All possible XOR combinations

can be represented by 2^16 values in a lookup table. For a row i of program memory, the XOR constraint is

$$E_1(\{w_{(i-1)m+1}, \ldots, w_{im}\}) = w_{(i-1)m+1} + 2^8 w_{(i-1)m+2} + \alpha w_{(i-1)m+3} + \beta + \gamma$$

where $w_{(i-1)m+1} = a$, $w_{(i-1)m+2} = b$, $w_{(i-1)m+3} = c$. Instead of directly mapping memory cells to values in $r_1, \ldots, r_e$, we map the output of the identities $E_1, \ldots, E_l$ to values in $r_1, \ldots, r_e$ instead.

We can use selector polynomials to toggle our randomized constraints on/off for a given row in program memory. This allows us to also mask out locations in program memory that are not table reads when required.

To validate the correctness of the values $\{r_1, \ldots, r_e\}$, we validate that the following permutation holds:

$$\frac{\prod_{i=1}^{n}(\sum_{j=1}^{l} E_j(\{w_{(i-1)m+1}, \ldots, w_{im}\}))}{\prod_{i=1}^{n}\prod_{j=1}^{s}(r_{(i-1)s+j} + q_{(i-1)s+j}\beta + \gamma)}$$

We can assume that, for any given $i \in [1, \ldots, n]$, only one randomized constraint will be active, which ensures that we are still evaluating a 'grand product'.

When the circuit is constructed, the number of table reads per sub-table will be known, which means that we can use a precomputed selector q to map each element of $\{r_1, \ldots, r_n\}$ to their respective sub-table.

Finally, we should note that each identity can map more than one cell in program memory into the lookup read table. For example, we could index a range table using the following constraint:

$$E_{range} = (w_1 + \beta^{range} + \gamma)(w_2 + \beta^{range} + \gamma)(w_3 + \beta^{range} + \gamma)$$

Because we have introduced a selector polynomial to toggle each constraint, each identity can only contain (for a width 4 program) cubic terms, without increasing the degree of PLONK's quotient polynomial $T(X)$.

4 From a sorted list to a lookup argument

We are now at the stage where we can validate that all of a program's table reads have been mapped into the set of values $R = \{r_1, \ldots, r_e\}$, split over s degree-n commitments.

We now need to provide the prover with a mechanism to remove duplicate elements. We can assume that the elements of R are sorted in monotonically increasing order (for each subset). If this is not the case, our final permutation argument will be rejected by the verifier.

We introduce s **mask** commitments, that contain elements $\{m_1, \ldots, m_e\}$. Where all elements take boolean values. N.B. there seems like there should be a way to not need this mask thingy, but it's extremely cheap to commit to using a Lagrange-base SRS.

For a given mask element, if $m_i = 1$, the prover is making a claim that two adjacent values in R are adjacent.

The verifier validate this by evaluating the following:

$$m_i^2 - m_i = 0$$

$$(r_{i-1} - r_i)m_i = 0$$

The latter identity always holds if $m_i = 0$. If $m_i = 1$, then $r_{i-1} = r_i$.

Note that this does not prevent the prover from *not* masking duplicates. This is not necessary because the final permutation argument will fail if duplicates have not been removed.

5 The final permutation check

We almost have a complete lookup argument. The final step is the validate that elements $\{r_1 m_1, \ldots, r_e m_e\}$ is a *subset* of the lookup table values $\{t_1, \ldots, t_k\}$.

We need to accommodate for situations where not all table elements have been read from in a given proof - a strict permutation won't suffice.

To this end, we introduce a second set of l masking polynomials $y_1(X), \ldots, y_l(X)$, containing elements $y_1, \ldots, y_k$.

We use a transition constraint to validate that

$$y_i^2 - y_i = 0$$

We represent our lookup table via l commitments to table keys, $\{tk_1(X), \ldots, tk_l(X)\}$, l commitments to table values $\{tv_1(X), \ldots, tv_l(X)\}$ and l commitments to subtable indices $\{ts_1(X), \ldots, ts_l(X)\}$.

The final permutation check is the following:

$$\frac{\prod_{i=1}^{n}\prod_{j=1}^{l}\left(y_{(i-1)l+j}(tk_{(i-1)l+j} + \alpha tv_{(i-1)l+j} + \beta ts_{(i-1)l+j} + \gamma) + (1 - y_{(i-1)l+j})\right)}{\prod_{i=1}^{n}\prod_{j=1}^{s}\left((1 - m_{(i-1)s+j})(r_{(i-1)s+j} + q_{(i-1)s+j}\beta + \gamma) + m_{(i-1)s+j}\right)}$$

6 Efficiency Analysis

Adding this lookup argument into PLONK incurs the following fixed overheads for the prover:

1. Two new grand product polynomials, to validate the two permutations (TODO: can these be combined somehow without exploding T(X)?)
2. A commitment to $M(X)$
3. A commitment to $Y(X)$

Assuming these are committed to using a Lagrange-base SRS, the number of scalar multiplications will be equal to the number of reads, for the grand product polynomials. Both $M(X)$ and $Y(X)$ can be committed to using group additions instead of group exponentiations, so their overall contribution to prover time is small.

We can also generously assume that less than 1/4 of all memory cells will be table reads. This sets $s = 1$. Similarly, we can likely assume that the set of all lookup table values will be less than n, which sets $l = 1$.

The final number of group exponentiations required by the prover for each table read is 3 - one to commit to $R(X)$, one for each new grand product polynomial.

7 Examples: XOR

We can perform a 32-bit XOR in 4 gates (using an 8bit xor and not the funky sparse bit trick), using a lookup table and one randomized constraint.

$$\begin{array}{cccc} a_0 & b_0 & c_0 & - - - \\ a_1 & b_1 & c_1 & - - - \\ a_2 & b_2 & c_2 & - - - \\ a_3 & b_3 & c_3 & - - - \end{array}$$

With the above structure, we can ensure that all of a_i, b_i, c_i track accumulating sums instead of raw 8-bit values - the randomized constraint can extract the 8-bit slices from the differences between elements. This means we don't have to swap between 8-bit representations and native representations of 32 bit integers, which makes it much easier to combine different arithmetic operations.

Total cost is 4 gates and 4 table reads. The 4 reads contribute 12 scalar multiplications. A gate's base 'cost' is 11 scalar multiplications, making the 32-bit XOR cost approximately 5 gates.

The current best case is the quaternary XOR custom gate, which requires 17 gates. This yields a ¿3x speed increase.

7.1 example: bit rotations on 32 bit integers

Assume we have a 32-bit unsigned integer, represented using 1 memory cell with value a. We wish to evaluate a bit rotation by a fixed number of bits. E.g. $a >>> 5$.

Any rotation by an odd number of bits will 'slice' one 16-bit value in two. In this case into a 11-bit value b_0, a 16-bit value b_1 and a 5-bit value b_2. i.e.

$$a >>> 5 = b_0 + 2^{16-5}b_1 + 2^{32-5}b_2$$

We can now validate the following:
$b = b_0 + 2^{16-5}b_1 + 2^{32-5}b_2$ $a = b_2 + 2^5 b_0 + 2^1 6 b_1$

This requires 3 table lookups, b_0 must come from an 11-bit range, b_1 from a 16-bit range, b_2 from a 5-bit range. Both of above conditions can be verified using a single gate (but would require the ability to validate 2 arithmetic statements per gate. Otherwise 2 gates).

Total cost is 1 gate and 3 reads, which is 2 native gates.

7.2 example: unsigned addition

Let's say we wish to evaluate $a + b = c$. Instead of using a basic add gate, we can use a lookup to ensure that c is within the range $[0, \ldots, 2^{32}]$. Specifically we validate:

$$a + b = c_0 + 2^{16}c_1 + 2^{32}c_2$$

We use a simple transition constraint to validate that $c_2 \in [0, 1]$. We use lookup table reads to validate that c_0 and c_1 are 16-bit values.

Like in the XOR and bit rotation case, we can represent c via accumulating sums in program memory, instead of raw 16-bit slices.

Total cost is 1 gate and 2 reads.

7.3 example: Blake2s

The Blake2s round function applies the following operation to three 32-bit variables a, b, c, where d is a constant rotation value:

$$e = (a + b)^c >>> d$$

By combining the above three sub-programs, we can evaluate this 'add xor rotate' operation in 4 gates and 4 table reads!

Layout in program memory is the following:

$$\left|\begin{array}{|c|c|c|c|} (a+b)_0 & c_0 & e_0 & - - - \\ (a+b)_1 & c_1 & e_1 & b \\ (a+b)_2 & c_2 & e_2 & (b + overflow) \\ (a+b)_3 & c_3 & e_3 & a \end{array}\right|$$

Step 1: all values 'a+b', 'c', 'e' track accumulating sums. So $(a+b)_3 = a+b \bmod 2^{32}$.
Step 2: we use a lookup table that combines an XOR with a bit rotation, instead of performing these in discrete steps.
Step 3: use a transition constraint that validates

$$a + b_3 = a + b + overflow$$

Step 4: use a transition constraint that validates

$$(b + overflow - b)^2 - (b + overlow - b) = 0$$

A Blake2s round consists of 4 of these, which is 16 gates and 16 table reads.

The blake2s hash function requires 80 applications of this round function, which would be 1,280 gates and 1,280 table reads. The table reads would cost 3,840 scalar multiplications, which is equivalent to 350 native gates. So the total cost would be $\approx 1,650$ 'gates'. Current best estimate with quaternary decomposition constraints is $\approx 6,700$. R1CS requires $> 20,000$.

8 RAM overview

The following describes a sub-protocol that emulates dynamic memory access within a PLONK circuit. Each RAM access has constant complexity and is updatable by the prover. The contrasts with the current PLONK memory model, which enables strict copy constraints between specific witnesses, that must be known at 'circuit construction time'.

A RAM 'triple' consists of the following three witnesses:

- An index $s \in F_p$
- A value $v \in F_p$
- A timestamp $t \in F_p$

Timestamp is 'when' in the execution trace a RAM lookup occurs (e.g. 1st lookup is $t_1 = 1$, 2nd is $t_2 = 2$ etc).

8.0.1 RAM memory access model

We model our RAM as write-after-read. That every RAM read is followed by a RAM write into the same index. Under this model, we can enable dynamic RAM by performing a consistency check - that every when a value is written into a RAM index, the subsequent access into that index will return the same value.

8.1 RAM consistency checks via reduced permutations

We define three sets of RAM triples:

- Set A: RAM triples sorted by timestamp
- Set B: RAM triples sorted by index, with triples of the same index sorted by timestamp
- Set C: the set of all RAM reads in program memory

Standard PLONK copy constraints are used to map between A and C, as the locations of specific RAM accesses within a circuit are fixed.

The verifier generates two random challenges $(\alpha, \beta) \in \mathbb{F}_p$, where for set elements $a_i \in A$:

$$a_i = t_i + \alpha \cdot s_{\sigma_t(i)} + \beta \cdot v_{\sigma_t(i)}$$

Here, $\sigma_t(i)$ is a prover-defined permutation that maps timestamp-ordered indices into index-ordered indices.

Similarly, for set elements $b_i \in B$:

$$b_i = t_{\sigma_s(i)} + \alpha \cdot s_i + \beta \cdot v_i$$

Where $\sigma_s(i)$ is a prover-defined permutation that maps index-ordered indices into timestamp-ordered indices.

A reduced permutation argument can be used to map between sets A and B. The verifier uses a grand-product argument to validate that

$$\prod_{i=1}^{n} \frac{(a_i + \gamma)}{(b_i + \gamma)}$$

8.2 RAM consistency checks via copy constraints

In set B, the verifier can use regular transition constraints to validate that adjacent triples have monotonically increasing indices. Specifically

$$(s_{i+1} - s_i)^2 - (s_{i+1} - s_i) = 0$$

This requires every RAM entry has a read/write multiplicity of at least one. Dummy reads can be added into a circuit to enable this.

A regular PLONK copy constraint can be used to validate that, for every pair of RAM triples, the index and value fields are identical.

$$\forall j \in [1, \ldots, \frac{n}{2}] : s_{2*j} = s_{2*j+1} \quad , \quad v_{2*j} = v_{2*j+1}$$

The final consistency check is to ensure that, for elements in B that share the same index, the timestamps are monotonically increasing. This validates that adjacent RAM accesses share identical values. i.e. every RAM read into a given index returns the previously written value.

8.3 RAM consistency checks via range checks

In set B, the set of witnesses $u = (t_{i+1} - t_i) \cdot (1 - (s_{i+1} - s_i))$ must fall within the range $0, \ldots, n$. Adjacent s indices are monotonically increasing by 0 or 1. i.e. if $s_{i+1} \neq s_i$, $u = 0$.

If $s_{i+1} = s_i$, $u = (t_{i+1} - t_i)$. i.e. if $u >= 0$, adjacent timestamp fields are monotonically increasing. The reduced permutation check between A and B ensures that $u \neq 0$ when $s_{i+1} = s_i$.

We can use our range protocol to validate that $u \in [0, \ldots, n]$.

8.4 Assigning RAM sets to program memory

For width-4 PLONK, a single row of program memory can contain two elements of set A. This is because we can add an implicit timestamp via selector polynomials. Layout in program memory is as follows:

$$\left| \begin{array}{c|c|c|c} s_{\sigma_t(i)} & v_{\sigma_t(i)} & s_{\sigma_t(i+1)} & v_{\sigma_t(i+1)} \\ \ldots & \ldots & \ldots & \ldots \\ s_{\sigma_t(n-1)} & v_{\sigma_t(n-1)} & s_{\sigma_t(n)} & v_{\sigma_t(n)} \end{array} \right|$$

Set B requires timestamps as explicit values in program memory, with one RAM entry occupying an entire row. Layout is as follows:

$$\left|\begin{array}{c|c|c|c} t_{\sigma_s(i)} & s_i & v_i & - - - \\ t_{\sigma_s(i+1)} & s_{i+1} & v_{i+1} & (t_{\sigma_s(i+1)} - t_{\sigma_s(i)}) \cdot (s_{i+1} - s_i) \\ \cdots & \cdots & \cdots & \cdots \\ t_{\sigma_s(n)} & s_n & v_n & (t_{\sigma_s(n)} - t_{\sigma_s(n-1)}) \cdot (s_n - s_{n-1}) \end{array}\right|$$

Set C will be dispersed across the remaining program memory cells. Strict copy constraints validate the mapping between C and A, therefore set C requires no explicit structure in program memory.

8.5 Permutation argument to map A to B

To verify our reduced permutation argument, we need to selectively integrate a subset of all witnesses into a grand product argument. To this end, we defined two selector polynomials $Q_{ram1}(X), Q_{ram2}(X)$, that used to conditionally include product terms into a grand product argument.

We include additional selector polynomials $Q_{t1}(X), Q_{t2}(X)$, that inject an implicit timestamp value into our timestamp-ordered set A. The permutation argument is as follows:

$$\begin{gathered} Z(Xw) \cdot \left(\begin{array}{c} Q_{ram1}(X)(Q_{t1}(X) + \alpha \cdot A(X) + \beta \cdot B(X) + \gamma) \\ \cdot (Q_{t2}(X) + \alpha \cdot C(X) + \beta \cdot D(X))) + (1 - Q_{ram1}(X)) \end{array} \right) \\ = \\ Z(X)\Big(Q_{ram2}(X)(A(X) + \alpha \cdot B(X) + \beta \cdot C(X) + \gamma) + (1 - Q_{ram2}(X))\Big) \bmod Z_H(X) \end{gathered}$$

One advantage of this construction is that $Z(X)$ can be considered sparse - adjacent elements do not change if a row of program memory is not a member of A or B. When computed using a Lagrange-base preprocessed SRS, a commitment to $Z(X)$ can be computed with n multi-exponentiations.

The final check we require, is that $Z(1) = 1$. This can be validated by validating that $(Z(X) - 1)L_1(X) = 0 \bmod Z_H(X)$.

8.6 Efficiency analysis

A RAM lookup will add 6 extra witnesses into program memory, across 1.5 rows. The lookup witnesses themselves require two values in program memory (the index and the value). The range check on u will require one further witness in program memory. This brings the overall cost of a RAM access to 2.25 PLONK gates.

9 Scheme without masks

Notation Fix integers n, d and vectors $a \in \mathbb{F}^n, t \in F^d$. We will (ab)use the notation $a \subset t$ to mean $\{a_i\}_{i \in [n]} \subset \{t_i\}_{i \in [d]}$.

When $a \subset t$, we say that a is *sorted by* t to mean that values appear in the same order in a as they do in t. Formally, for any $i < i' \in [n]$ such that $a_i \neq a_{i'}$, if $j, j' \in [d]$ are such that $t_j = a_i, t_{j'} = a_{i'}$ then $j < j'$.

Now, given $t \in \mathbb{F}^d, a \in \mathbb{F}^n, s \in \mathbb{F}^{n+d}$, define bi-variate polynomials F, G as

$$F(\beta, \gamma) := (1+\beta)^n \cdot \prod_{i \in [n]} (\gamma + a_i) \prod_{i \in [d-1]} (\gamma(1+\beta) + t_i + \beta t_{i+1})$$

$$G(\beta, \gamma) := \prod_{i \in [n+d-1]} (\gamma(1+\beta) + s_i + \beta s_{i+1})$$

we have

Claim 9.1. *$F \equiv G$ if and only if*

1. *$a \subset t$, and*
2. *s is (a, t) sorted by t.*

Proof. We first write f, g as elements of $\mathbb{F}(\beta)[\gamma]$ while taking out a $(1+\beta)$ factor as follows.

$$F(\beta, \gamma) = (1+\beta)^{n+d-1} \cdot \prod_{i \in [n]} (\gamma + a_i) \prod_{i \in [d-1]} (\gamma + (t_i + \beta t_{i+1})/(1+\beta))$$

$$G(\beta, \gamma) = (1+\beta)^{n+d-1} \prod_{i \in [n+d-1]} (\gamma + (s_i + \beta s_{i+1})/(1+\beta))$$

Suppose that $a \subset t$ and $s \in \mathbb{F}^{n+d}$ is (a, t) sorted by t.

Then for each $j \in [d-1]$, there is an index $i \in [n+d-1]$ where, such that $(t_j, t_{j+1}) = (s_i, s_{i+1})$. The corresponding factors in F, G are equal. That is,

$$(\gamma + (t_i + \beta t_{i+1})/(1+\beta)) = (\gamma + (t_i + \beta t_{i+1})/(1+\beta))$$

For the other direction, assume $F \equiv G$ as polynomials in $\mathbb{F}[\beta, \gamma]$ Then $F \equiv G$ also as elements of $\mathbb{F}(\beta)[\gamma]$. Since $\mathbb{F}(\beta)[\gamma]$ is a unique factorization domain, we know that the linear factors of F, G, as written above must be equal. Thus, for each $i \in [d-1]$, G must have a factor equal to $(\gamma + (t_i + \beta t_{i+1})/(1+\beta))$. In other words, for some $j \in [n+d-1]$,

$$\gamma + (t_i + \beta t_{i+1})/(1+\beta) = \gamma + (s_j + \beta s_{j+1})/(1+\beta),$$

which implies $t_i + \beta t_{i+1} = s_j + \beta s_{j+1}$, and therefore $t_i = s_j, t_{i+1} = s_{j+1}$. Call $P \subset [n+d-1]$ the set of these $d-1$ indices j. For any other index $j \in [n+d-1] \setminus P$, there must be a factor "coming from a" in F that equals the corresponding factor in G. More precisely, for such j there exists $i \in [n]$ such that

$$\gamma + a_i = \gamma + (s_j + \beta s_{j+1})/(1+\beta),$$

or equivalently

$$a_i + \beta a_i = s_j + \beta s_{j+1}$$

which implies $a_i = s_j = s_{j+1}$.

Thus, we know that whenever consecutive values in s are different, they are exactly equal to two consecutive values in t, and all values of a are values of t.

□

Claim 9.1 motivates the following protocol. Let $H = \{\mathbf{g}, \ldots, \mathbf{g}^{n+d}\}$, $H_1 := \{\mathbf{g}, \ldots, \mathbf{g}^{n}\}$, $H_2 := \{\mathbf{g}^{n+1}, \ldots \mathbf{g}^{n+d}\}$

Preprocessed polynomials: The polynomial $t \in \mathbb{F}_{<d}[X]$ describing the lookup values .

Inputs: $f \in \mathbb{F}_{<n}[X]$

Protocol:

1. $\mathrm{P}_{\mathsf{poly}}$ computes a polynomial $s \in \mathbb{F}_{<n+d}[X]$ that, identifying polynomials with vectors in $\mathbb{F}^{n+d}$ according to their values on H, is (f, t) sorted by t.
2. $\mathrm{P}_{\mathsf{poly}}$ sends s to $\mathcal{I}$.
3. $\mathrm{V}_{\mathsf{poly}}$ chooses random $\beta, \gamma \in \mathbb{F}$ and sends them to $\mathrm{P}_{\mathsf{poly}}$.
4. $\mathrm{P}_{\mathsf{poly}}$ computes a polynomial $Z \in \mathbb{F}_{<n+d}[X]$ that aggregates the value $F(\beta, \gamma)/G(\beta, \gamma)$ where F, G are as described above. Specifically, we let
   (a) $Z(\mathbf{g}) = 1$,
   (b) $Z(\mathbf{g}^i) = \prod_{1 \le j < i}(\gamma + f(\mathbf{g}^j))/(\gamma(1+\beta) + s(\mathbf{g}^j) + \beta s(\mathbf{g}^{j+1}))$, for $2 \le i \le n+1$
   (c)

$$Z(\mathbf{g}^{n+i}) = \prod_{j \in [n]} (\gamma + f_j) \cdot \prod_{1 \le j < i} (\gamma(1+\beta)t_j + \beta t_{j+1}) / \left(\prod_{1 \le j < n+i} (\gamma(1+\beta)s_j + \beta s_{j+1}) \right).$$

   and
5. $\mathrm{P}_{\mathsf{poly}}$ sends Z to $\mathcal{I}$.
6. $\mathrm{V}_{\mathsf{poly}}$ checks that Z is of the form described similarly to the PLONK perm argument and that $Z(\mathbf{g}^{n+d}) = 1$. (TODO:write exact ver equations) and outputs acc iff all checks hold.

Lemma 9.2. *Suppose that $\{f(\mathbf{g}^i)\}_{i \in [n]}$ is not contained in $\{t(\mathbf{g}^i\}_{i \in [d]}$. Then for any strategy of $\mathcal{A}$ playing the role of $\mathrm{P}_{\mathsf{poly}}$ in the above protocol, the probability that $\mathrm{V}_{\mathsf{poly}}$ accepts is $\mathsf{negl}(\lambda)$.*

Proof. Using Claim 9.1 we know that when f's range is not contained in t's for any choice of s sent by $\mathrm{P}_{\mathsf{poly}}$ the polynomials $F(\beta,\gamma), G(\beta,\gamma)$ are different. From the SZ lemma e.w.p $\mathsf{negl}(\lambda)$ $\mathrm{V}_{\mathsf{poly}}$ chooses β,γ such that $F(\beta,\gamma) \neq G(\beta,\gamma)$. In this case we have $Z(\mathbf{g}^{n+d}) \neq 1$ which means $\mathrm{V}_{\mathsf{poly}}$ rejects. □