Lesson 18

Week 5

Lesson 17 - STARK implementation

Lesson 18 - Plonk part 1 / Linea

Lesson 19 - Plonk part 2 / Boojum

Lesson 20 - ZKML

Today's topics

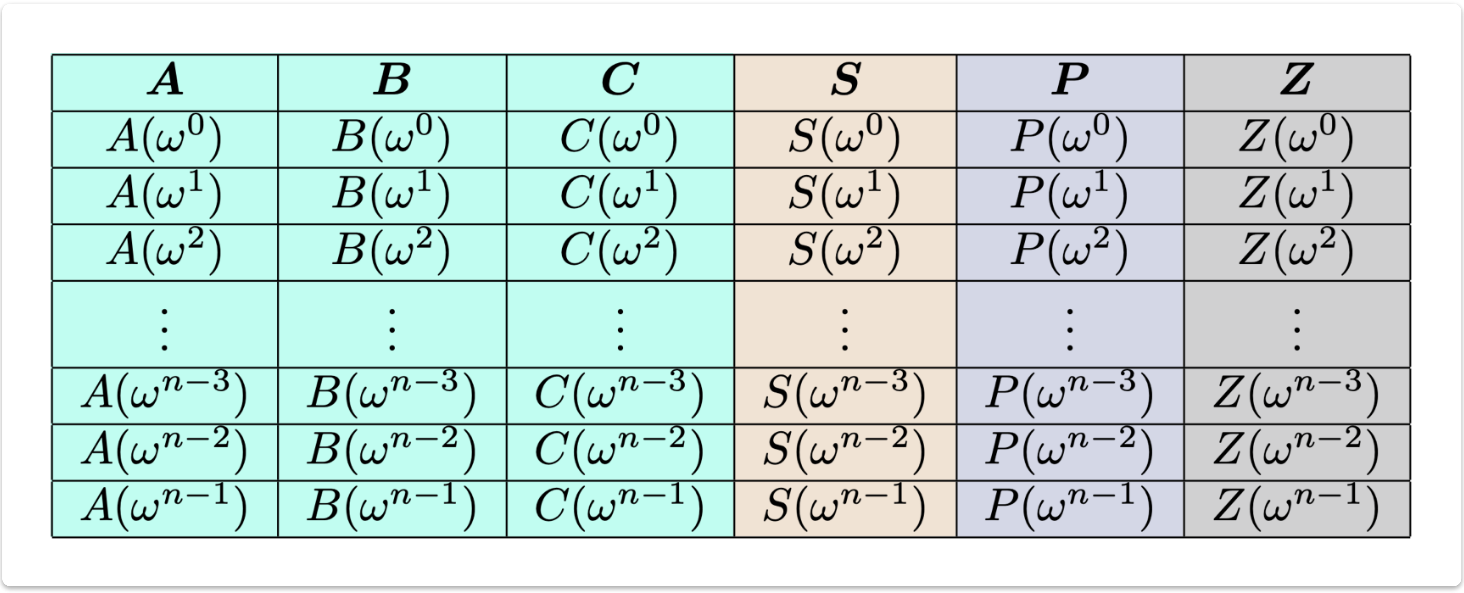
- Arithmetisation review / STARK Arithmetisation continued.
- Trusted Setup
- Circuit / QAP process in PLONK
- Plonkish protocols
- Future directions

Arithmetisation review

We have seen arithmetisation in various proving systems, and their use of tables makes them look similar.
SNARKs arithmetisation is based on arithmetic circuits, composed of addition and multiplication gates (and custom gates), from which R1CS are built to produce vectors and then a QAP to produce polynomials.
With STARKs we have boundary and transition constraints, from which we can create polynomials.

When implementing these systems it is convenient to use tables which represent gates, or steps in a virtual machine.

From Scroll article



$\boldsymbol{A}$	$\boldsymbol{B}$	$\boldsymbol{C}$	$\boldsymbol{S}$	$\boldsymbol{P}$	$\boldsymbol{Z}$
$A(\omega^0)$	$B(\omega^0)$	$C(\omega^0)$	$S(\omega^0)$	$P(\omega^0)$	$Z(\omega^0)$
$A(\omega^1)$	$B(\omega^1)$	$C(\omega^1)$	$S(\omega^1)$	$P(\omega^1)$	$Z(\omega^1)$
$A(\omega^2)$	$B(\omega^2)$	$C(\omega^2)$	$S(\omega^2)$	$P(\omega^2)$	$Z(\omega^2)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$A(\omega^{n-3})$	$B(\omega^{n-3})$	$C(\omega^{n-3})$	$S(\omega^{n-3})$	$P(\omega^{n-3})$	$Z(\omega^{n-3})$
$A(\omega^{n-2})$	$B(\omega^{n-2})$	$C(\omega^{n-2})$	$S(\omega^{n-2})$	$P(\omega^{n-2})$	$Z(\omega^{n-2})$
$A(\omega^{n-1})$	$B(\omega^{n-1})$	$C(\omega^{n-1})$	$S(\omega^{n-1})$	$P(\omega^{n-1})$	$Z(\omega^{n-1})$

STARK Arithmetisation continued

> **A more complex example**
>
> See article
> Imagine our code calculates the first 512 Fibonacci sequence
> 1,1,2,3,5 ...

If we decide to operate on a finite field with max number 96769
And we have calculated that the 512th number is 62215.

Then our constraints are

$$A_{0,2} - 1 = 0$$
$$A_{1,2} - 1 = 0$$

$$\forall 0 >= i <= 510 : A_{i+2,2} = A_{i+1,2} + A_{i,2}$$

$$A_{511,2} - 62215 = 0$$

Creating a polynomial for our trace

In order to efficiently prove the validity of the execution trace, we strive to achieve the following two goals:

1. Compose the constraints on top of the trace polynomials to enforce them on the trace.
2. Combine the constraints into a single (larger) polynomial, called the Composition Polynomial, so that a single low degree test can be used to attest to their low degree.

We define a polynomial $f(x)$ such that the elements in the execution trace are evaluations of f in powers of some generator g.
Recall our finite field will have generators, we use these to index the steps of our trace.

Taking the fibonacci example from the medium article
we can create constraints such as

$$\forall\, x \in \{1, g^2, g^3 \ldots g^{509}\}: f(g^2x) - f(gx) - f(x) = 0$$

this constrains the values between subsequent rows.
It also means that the g values are roots of this polynomial.

We can therefore use the approach we saw earlier to provide the vanishing polynomial by using the term
$(x - g^i)$

and from this we create the *composition polynomial*

$q(x) = \frac{f(g^2x)-f(gx)-f(x)}{\prod_{i=0}^{509}(x-g^i)}$

from the basic fact about polynomials and their roots is that
if $p(x)$ is a polynomial, then
$p(a) = 0$ for some specific value a,
if and only if there exists a polynomial $q(x)$
such that
$(x - a)q(x) = p(x)$, and
$deg(p) = deg(q) + 1$.

This expression agrees with the polynomial of degree at most 2 if our execution trace has been correct, i.e obeyed the step constraint that we defined.
If the trace differs from that, then this expression would be unlikely to produce a low degree polynomial.

Creating a polynomial for our trace

In order to efficiently prove the validity of the execution trace, we strive to achieve the following two goals:

1. Compose the constraints on top of the trace polynomials to enforce them on the trace.
2. Combine the constraints into a single (larger) polynomial, called the Composition Polynomial, so that a single low degree test can be used to attest to their low degree.

We define a polynomial $f(x)$ such that the elements in the execution trace are evaluations of f in powers of some generator g.
Recall our finite field will have generators, we use these to index the steps of our trace.
Taking the fibonacci example from the medium article
we can create constraints such as

$$\forall\, x \in \{1, g^2, g^3 \ldots g^{509}\}\text{: } f(g^2x) - f(gx) - f(x) = 0$$

this constrains the values between subsequent rows.
It also means that the g values are roots of this polynomial.

We can therefore use the approach we saw earlier to provide the vanishing polynomial by using the term
$(x - g^i)$

and from this we create the *composition polynomial*

$q(x) = \frac{f(g^2x)-f(gx)-f(x)}{\prod_{i=0}^{509}(x-g^i)}$

from the basic fact about polynomials and their roots is that
if $p(x)$ is a polynomial, then
$p(a) = 0$ for some specific value a,
if and only if there exists a polynomial $q(x)$
such that
$(x - a)q(x) = p(x)$, and
$deg(p) = deg(q) + 1.$

This expression agrees with the polynomial of degree at most 2 if our execution trace has been correct, i.e obeyed the step constraint that we defined.
If the trace differs from that, then this expression would be unlikely to produce a low degree polynomial.

PLONK

From Zac Williamson

"PLONK's strength is these things we call custom gates. It's basically you can define your own custom bit arithmetic operations. I guess you can call them mini gadgets.
That are extremely efficient to evaluate inside a PLONK circuit.
So you can do things like do elliptical curve point addition in a gate.
You can do things like efficient Poseidon hashes, Pedersen hashes. You can do parts of a SHA-256 hash.
You can do things like 8-bit logical XOR operations.
All these like explicit instructions which are needed for real world circuits, but they're all quite custom."

> ### SNARK Trusted Setup
>
> See overview
>
> #### Powers of Tau
>
> From the overview
> Consider two polynomials, A and B, each of degree $n = 2^{18} - 1$. The prover seeks to multiply these two polynomials, obtaining a resulting polynomial, P.
> Multiplying two polynomials of degree n will produce a polynomial of degree $2n$.
> In this instance, after the multiplication, the degree of P

would be $2 \times (2^{18} - 1)$. Subsequently, P must be evaluated at a random value, denoted as τ (tau), without revealing the coefficients of P to a verifier.

This process forms a commitment to polynomials A and B because the prover demonstrates the ability to evaluate P at τ without revealing A, B, or P explicitly. Verifiability is achieved through the use of cryptographic techniques to assure the verifier that $P(\tau)$ is computed correctly without conveying any additional information about P, A, or B.

Process

1. From a field $\mathbb{F}_\mathbb{p}$ we get a random point s
2. We then create $\{G, sG, s^2G, \ldots\}$ up to the degree of the polynomials we are expecting.
   This is our structured reference string
3. The s value is the 'toxic waste' and is deleted
4. The prover will need to evaluate $f(s)$ later but doesn't know s, but does have the s values 'hidden' in the SRS

Creating the commitment

So if our original polynomial is $f(x)$ = $f_0 + f_1x + f_2x^2 + \ldots$ and the prover wants to evaluate this at s

We can rewrite our SRS as

$\{G, sG, s^2G, \ldots\}$ = $\{\tau_0, \tau_1, \tau_2, \ldots\}$

then we want to rewrite $f(s)$ in terms of $f_0\tau_0 + f_1\tau_1 + f_2\tau_2 + \ldots$

= $f_0G + f_1sG + f_2s^2G$

$= (f_0 + f_1s + f_2s^2 + \ldots)G$

$= f(s)G$

this is our commitment to f

$= com(f)$

so our commitment is hidden behind the group element G, which is a generator of an elliptic curve group and hence $f(s)G$ is a curve point.

Circuit / QAP theory in PLONK

The Plonk constraint system is similar to the rank-one constraint system (R1CS) in that they both can have only one multiplication in each gate. The difference is that R1CS allows unlimited additions in a gate, while Plonk can only do one, excluding the addition of a constant

Given a program P, you convert it into a circuit, and generate a set of equations that look like this:
Each equation is of the following form (L = left, R = right, O = output, M = multiplication, C = constant):
and a_i, b_i are the wire values

What the prover and verifier can calculate ahead of time

The program-specific polynomials that the prover and verifier need to compute ahead of time are:
$Q_L(x), Q_R(x), Q_O(x), Q_M(x), Q_C(x)$, which together represent the gates in the circuit
(note that $Q_C(x)$ encodes public inputs, so it may need to be computed or modified at runtime)

The "permutation polynomials" $\sigma_a(x), \sigma_b(x)$ and $\sigma_c(x)$, which encode the copy constraints between the a, b,and c wires

We can write constraints of the form :

$(Q_{L_i})a_i + (Q_{R_i})b_i + (Q_{O_i})c_i + (Q_{M_i})a_ib_i + Q_{C_i} = 0$

You then convert this set of equations into a single polynomial equation:
$Q_L(x)a(x) + Q_R(x)b(x) + Q_O(x)c(x) + Q_M(x)a(x)b(x) +$

You also generate from the circuit a list of copy constraints. From these copy constraints you generate the three polynomials representing the permuted wire indices:
$\sigma_a(x), \sigma_b(x), \sigma_c(x)$

To generate a proof, you compute the values of all the wires and convert them into three polynomials:
$a(x), b(x), c(x)$
You also compute six "coordinate pair accumulator" polynomials as part of the permutation-check argument.

Finally you compute the cofactors $H_i(x)$

There is a set of equations between the polynomials that need to be checked; you can do this by making commitments to the polynomials, opening them at some random z (along with proofs that the openings are correct), and running the equations on these evaluations instead of the original polynomials.
The proof itself is just a few commitments and openings and can be checked with a few equations.

Implementation details of Plonkish circuits

From Halo2 Book

PLONKish circuits are defined in terms of a rectangular matrix of values. We refer to *rows*, *columns*, and *cells* of this matrix with the conventional meanings.

A PLONKish circuit depends on a *configuration*:

- A finite field F, where cell values (for a given statement and witness) will be elements of F.
- The number of columns in the matrix, and a specification of each column as being *fixed*, *advice*, or *instance*. Fixed columns are fixed by the circuit; advice columns correspond to witness values; and instance columns are normally used for public inputs (technically, they can be used for any elements shared between the prover and verifier).
- A subset of the columns that can participate in equality constraints.
- A *maximum constraint degree*.
- A sequence of *polynomial constraints*. These are multivariate polynomials over F that must evaluate to zero *for each row*. The variables in a polynomial constraint may refer to a cell in a given column of the current row, or a given column of another row relative to this one (with wrap-around, i.e. taken modulo n). The maximum degree of each polynomial is given by the maximum constraint degree.

- A sequence of *lookup arguments* defined over tuples of *input expressions* (which are multivariate polynomials as above) and *table columns*.

A PLONKish circuit also defines:

- The number of rows n in the matrix. n must correspond to the size of a multiplicative subgroup of F×; typically a power of two.
- A sequence of *equality constraints*, which specify that two given cells must have equal values.
- The values of the fixed columns at each row.

From a circuit description we can generate a *proving key* and a *verification key*, which are needed for the operations of proving and verification for that circuit.

Note that we specify the ordering of columns, polynomial constraints, lookup arguments, and equality constraints, even though these do not affect the meaning of the circuit. This makes it easier to define the generation of proving and verification keys as a deterministic process.

Typically, a configuration will define polynomial constraints that are switched off and on by *selectors* defined in fixed columns.
For example, a constraint $q_i \cdot p(\ldots) = 0$ can be switched off for a particular row i by setting $q_i = 0$

> In this case we sometimes refer to a set of constraints controlled by a set of selector columns that are designed to be used together, as a *gate*.
> Typically there will be a *standard gate* that supports generic operations like field multiplication and division, and possibly also *custom gates* that support more specialised operations.

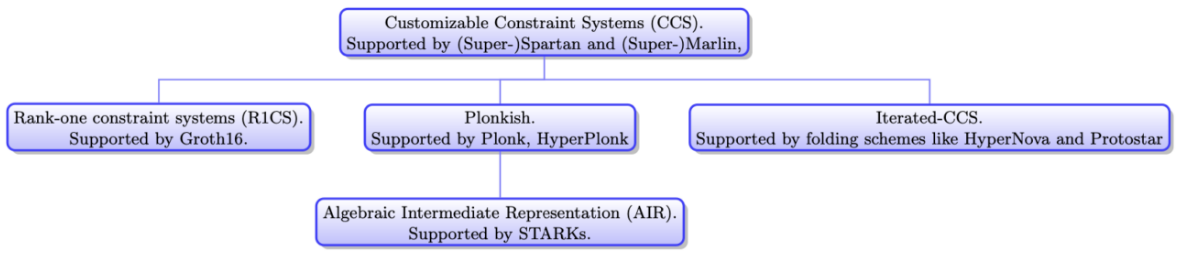
Plonkish protocols

(fflonk, turbo PLONK, ultra PLONK, hyperplonk , plonkup, plonky2 etc.)

Overview of types

Customisable Constraint Systems

See Paper
"Customizable constraint system (CCS) is a generalization of R1CS that can simultaneously capture R1CS, Plonkish, and AIR without overheads.
Unlike existing descriptions of Plonkish and AIR, CCS is not tied to any particular proof system."
CCS witnesses tend to be smaller than Plonkish ones.



Customizable Constraint Systems (CCS). Supported by (Super-)Spartan and (Super-)Marlin,
Rank-one constraint systems (R1CS). Supported by Groth16.
Plonkish. Supported by Plonk, HyperPlonk
Iterated-CCS. Supported by folding schemes like HyperNova and Protostar
Algebraic Intermediate Representation (AIR). Supported by STARKs.

Figure 1: Intermediate representations (i.e., kinds of circuits) and the back-ends that can prove statements about them. The top-most box is the most general kind of circuit amongst those depicted, and each child is a special case of its parent. A subtlety not portrayed in the depicted taxonomy is that some SNARKs (such as Spartan and STARKs) can avoid having an honest party pre-process circuits if the circuits have repeated structure, while others cannot (Groth16, Marlin, Plonk).

Plonky2

From article

Plonky2 also allows us to speed up proving times for proofs that don’t involve recursion. With FRI, you can either have fast proofs that are big (so they're more expensive to verify on Ethereum), or you can have slow proofs that are small. Constructions that use FRI, like the STARKs that Starkware uses in their ZK-rollups, have to choose; they can't have maximally fast proving times and proof sizes that are small enough to reasonably verify on Ethereum.

Plonky2 eliminates this tradeoff. In cases where proving time matters, we can optimise for maximally fast proofs. When these proofs are recursively aggregated, we're left with a single proof that can be verified in a small circuit. At this point, we can optimize for proof size.
We can shrink our proof sizes down to 45kb with only 20s of proving time (not a big deal since we only generate when we submit to Ethereum).

Plonky2 is natively compatible with Ethereum. Plonky2 requires only keccak-256 to verify a proof. We've estimated that the gas cost to verify a plonky2 size-optimized proof on Ethereum will be approximately 1 million gas.

However, this cost is dominated by the CALLDATA costs to publish the proof on Ethereum.
Since CALLDATA was repriced in EIP-4488, the verification cost of a plonky2 proof has dropped to between 170-200k gas, which could make it not only the

> fastest proving system, but also the cheapest to verify on Ethereum.
>
> **Other plonkish protocols**
>
> Halo2 - uses IPA for PCS
>
> Scroll - uses KZG

Overview of costs

Taken from Scroll article

- Phase 1: Filling in the trace table
  - Filling in witness data requires arithmetic operations over a large finite field
  - Trace tables are generally very large, due to a blow up factor when converting complex computation to a table/circuit format
  - Auxiliary columns require additional arithmetic operations and sorting
- Phase 2: Committing to the trace table
  - Committing to each column requires a size n MSM
- Phase 3: Proving the trace table's correctness
  - Computing the quotient polynomial's evaluation form requires
    - A size n iFFT for each column
    - A size 2n FFT for each column
    - Arithmetic operations over a large finite field
  - Committing to the quotient polynomial requires
    - A size 2n iFFT to convert to evaluation form
    - 2 size n MSMs, one per split polynomial
  - Generating the KZG evaluation proofs requires

- A size n MSM for each column
- 2 size n MSMs for the (split) quotient polynomial

The future direction of Plonkish protocols

See talk by Aztec at Devcon VI

From PLONK to PLONK-ish

PLONK

- constant size proof
- fast verification time
- universal trusted setup

PLONK-ish

- custom gates
- lookup protocols
- range checks
- DSLs + tooling

Next generation

- linear time prover
- sumcheck
- CISC style circuits
- more recursion

Comparison with sumcheck

	Standard	Sumcheck
Interpolation	Univariate	Multilinear
Proof Size	O(1)	O(log n)
Verification time	O(1)	O (log n)

	Standard	Sumcheck
Proving time	O(n log n)	O(n)

Plonk Resources

See PLONK paper

Also see Understanding PLONK by Vitalik

Plonk series

- Part 1
- Part 2
- Part 3
- Part 4
- Part 5