geometry

ZKP Languages: Where We Are Now

NOV | 22

ZKP Languages: Where We Are Now

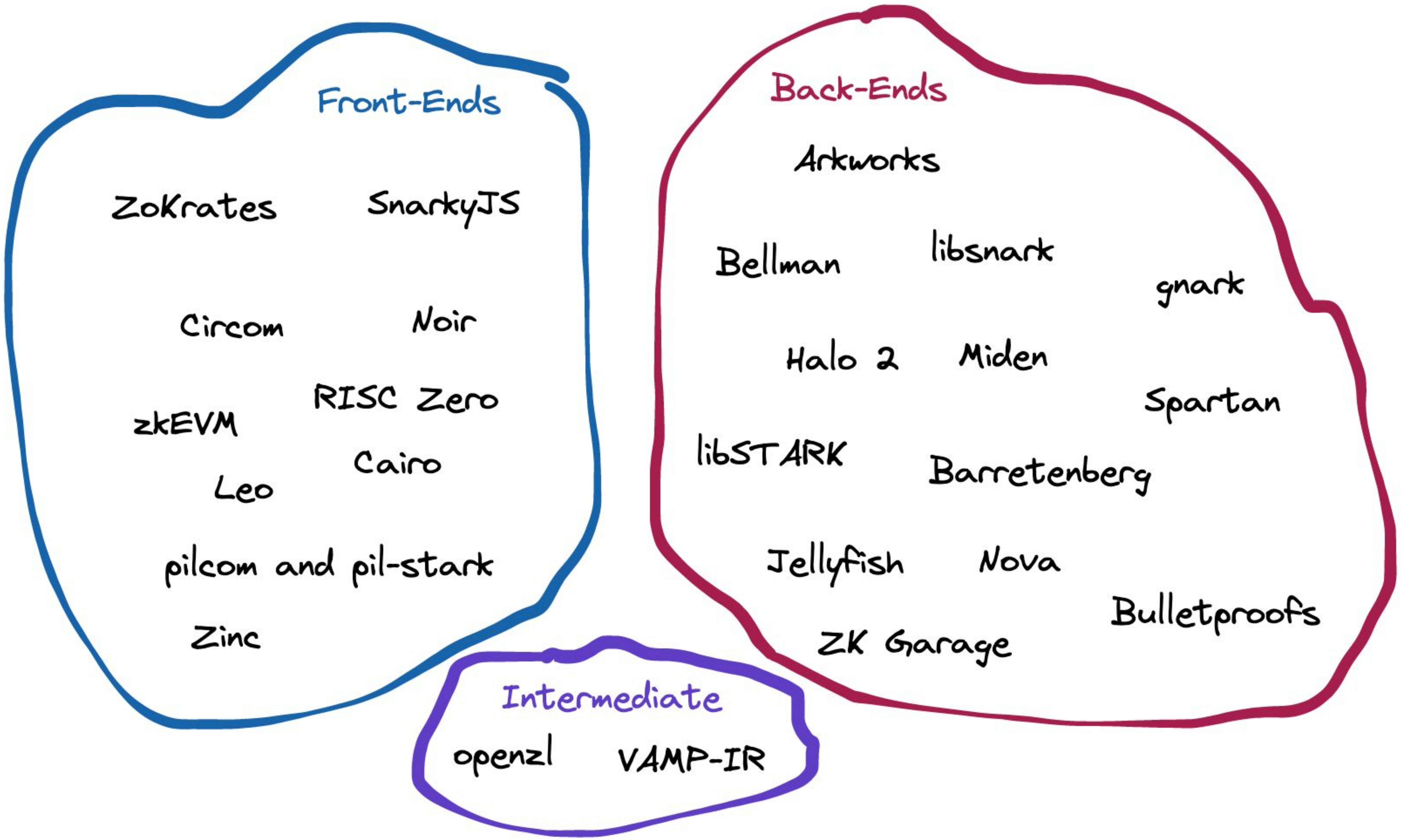
- Developer experience
- “Deployment ready”
- What we can use now
- What is practical in the future
- NOT what is theoretically possible

Brief history of ZK languages

Brief history of ZK languages

ZoKrates
SnarkyJS
Arkworks
Bellman
libsnark
gnark
Circom
Noir
Halo 2
Miden
RISC Zero
zkEVM
Spartan
libSTARK
Cairo
Barretenberg
Leo
pilcom and pil-stark
Jellyfish
Nova
Zinc
Bulletproofs
ZK Garage
openzl
VAMP-IR

Brief history of ZK languages



Brief history of ZK languages

$$\left(\sum a_i z_i\right)\left(\sum b_i z_i\right) = \sum c_i z_i$$

Brief history of ZK languages

- ZoKrates
- Bellman and Circom
- libSTARK

Brief history of ZK languages

- Marlin - R1CS
- Plonk - new arithmetization
- ZKVMs
- New ecosystems

Evolution of ZK languages

- Ergonomics
- “Magic”
- Safety
- Target audience
- Performance
- Ecosystem

libsnark

```
protoboard<FieldT> pb;

pb_variable<FieldT> A, B, less, less_or_eq;
A.allocate(pb, "A");
```

```
this->pb.add_r1cs_constraint(r1cs_constraint<FieldT>(1, (FieldT(2)^n) + B
```

```
pb.val(A[k]) = (i & (1ul<<k) ? FieldT::one() : FieldT::zero());
```

ZoKrates

```
def main(a, b, c):
  return a * b * c
```

Bellman

```
fn mixing_g<Scalar: PrimeField, CS: ConstraintSystem<Scalar>, M>(
    mut cs: M,
    v: &mut [UInt32],
    (a, b, c, d): (usize, usize, usize, usize),
    x: &UInt32,
    y: &UInt32,
) -> Result<(), SynthesisError>
where
    M: ConstraintSystem<Scalar, Root = MultiEq<Scalar, CS>>,
{
    v[a] = UInt32::addmany(
        cs.namespace(|| "mixing step 1"),
        &[v[a].clone(), v[b].clone(), x.clone()],
    )?;
    v[d] = v[d].xor(cs.namespace(|| "mixing step 2"), &v[a])?.rotr(R1);
    v[c] = UInt32::addmany(
        cs.namespace(|| "mixing step 3"),
        &[v[c].clone(), v[d].clone()],
    )?;
```

circom

```
template MerkleTreeChecker(levels) {
    signal input leaf;
    signal input root;
    signal input pathElements[levels];
    signal input pathIndices[levels];

    component selectors[levels];
    component hashers[levels];

    for (var i = 0; i < levels; i++) {
        selectors[i] = DualMux();
        selectors[i].in[0] <== i == 0 ? leaf : hashers[i - 1].hash;
        selectors[i].in[1] <== pathElements[i];
        selectors[i].s <== pathIndices[i];

        hashers[i] = HashLeftRight();
        hashers[i].left <== selectors[i].out[0];
        hashers[i].right <== selectors[i].out[1];
    }

    root === hashers[levels - 1].hash;
}
```

Arkworks

```
let aggregate_pk = G2Var::conditionally_select(
    &index_bit,
    &constrained_epoch.aggregate_pk,
    &dummy_pk,
)?;

let prepared_aggregate_pk = PairingVar::prepare_g2(&aggregate_pk)?;

let message_hash = G1Var::conditionally_select(
    &index_bit,
    &constrained_epoch.message_hash,
    &dummy_message,
)?;
```

Leo

```
transition mint_public(
    public receiver: address,
    public amount: u64,
) -> token {
    async finalize(receiver, amount);
    return token {
        owner: receiver,
        gates: 0u64,
        amount,
    };
}

finalize mint_public(
    public receiver: address,
    public amount: u64,
) {
    increment(balances, receiver, amount);
}
```

ZKVMs

- RISC Zero
- Cairo
- Miden
- zkEVM

RISC Zero

```rust
let params: RoundParams = env::read();
let result = params.process();
env::write(&result);
env::commit(&RoundCommit {
    old_state: *sha::digest(&params.state),
    new_state: *sha::digest(&result.state),
    shot: params.shot,
    hit: result.hit,
});
```

```rust
for i in 0..NUM_SHIPS {
    let ship = &mut state.ships[i];
    let span = SHIP_SPANS[i] as u32;
    let x = ship.pos.x;
    let y = ship.pos.y;
    let hit_shift = match ship.dir {
        ShipDirection::Horizontal => {
            if shot.y == y && shot.x >= x && shot.x < x + span {
                HitShift::Hit(shot.x - x)
            } else {
                HitShift::Miss
            }
        }
```

Cairo

```
let (local key) = voprf_key.read()
let (commitment) = commitments.read(user)
local commitment = commitment
let (t_hash) = hash2{hash_ptr=pedersen_ptr}(user, 0)
assert t_hash_y * t_hash_y = t_hash * t_hash * t_hash + t_hash + 3141592653589793238462643383279502884197169399375105820974944592307816406665
```

Miden

```
define MiMC over prime field (${modulus}) {

    const alpha: 3;

    static roundConstant: cycle prng(sha256, 0x4d694d43, 64);

    secret input startValue: element[1];

    // transition function definition
    transition 1 register {
        for each (startValue) {
            init { yield startValue; }

            for steps [1..${steps - 1}] {
                yield $r0^3 + roundConstant;
            }
        }
    }

    // transition constraint definition
    enforce 1 constraint {
        for all steps {
            enforce transition($r) = $n;
        }
    }
```

zkEVM

```
function name() public view returns (string)
function symbol() public view returns (string)
function decimals() public view returns (uint8)
function totalSupply() public view returns (uint256)
function balanceOf(address _owner) public view returns (
function transfer(address _to, uint256 _value) public re
function transferFrom(address _from, address _to, uint25
function approve(address _spender, uint256 _value) publi
function allowance(address _owner, address _spender) pub
```

PLONK languages

- Extensible frameworks - Noir, openzl, plicom/pil-stark, halo 2
- Good implementations of PLONK with a specific IOP - barretenberg, jellyfish

$$
\begin{aligned}
&\mathsf{t}(X) = \\
&\quad (\mathsf{a}(X)\mathsf{b}(X)\mathsf{q_M}(X) + \mathsf{a}(X)\mathsf{q_L}(X) + \mathsf{b}(X)\mathsf{q_R}(X) + \mathsf{c}(X)\mathsf{q_O}(X) + \mathsf{PI}(X) + \mathsf{q_C}(X))\,\frac{1}{\mathsf{Z_H}(X)} \\
&+ ((\mathsf{a}(X) + \beta X + \gamma)(\mathsf{b}(X) + \beta k_1 X + \gamma)(\mathsf{c}(X) + \beta k_2 X + \gamma)\mathsf{z}(X))\,\frac{\alpha}{\mathsf{Z_H}(X)} \\
&- ((\mathsf{a}(X) + \beta \mathsf{S}_{\sigma 1}(X) + \gamma)(\mathsf{b}(X) + \beta \mathsf{S}_{\sigma 2}(X) + \gamma)(\mathsf{c}(X) + \beta \mathsf{S}_{\sigma 3}(X) + \gamma)\mathsf{z}(X\omega))\,\frac{\alpha}{\mathsf{Z_H}(X)} \\
&+ (\mathsf{z}(X) - 1)\,\mathsf{L}_1(X)\,\frac{\alpha^2}{\mathsf{Z_H}(X)}
\end{aligned}
$$

Vamp-IR

```
def twisted_edwards_add[A, D] x1 y1 x2 y2 -> x3 y3 {
  (1 + D*x1*x2*y1*y2)*x3 = x1*y2 + y1*x2
  (1 - D*x1*x2*y1*y2)*y3 = y1*y2 - A*x1*x2
}
```

SnarkyJS

```
class Main extends Circuit {
  @circuitMain
  static main(preimage: Field, @public_ hash: Field) {
    Poseidon.hash([preimage]).assertEquals(hash);
  }
}
```

```
class NotSoSimpleZkapp extends SmartContract {
  @state(Field) x = State<Field>();

  @method init(proof: TrivialProof) {
    proof.verify();
    this.x.set(Field(1));
  }

  @method update(
    y: Field,
    oldProof: SelfProof<ZkappPublicInput>,
    trivialProof: TrivialProof
  ) {
    oldProof.verify();
    trivialProof.verify();
    let x = this.x.get();
    this.x.assertEquals(x);
    this.x.set(x.add(y));
  }
}
```

Noir

```
#[foreign(merkle_membership)]
fn check_membership(_root : Field, _leaf : Field, _index : Field, _hash_path: [Field]) -> Field {}


// Returns the root of the tree from the provided leaf and its hashpath, using pedersen hash
fn compute_root_from_leaf(leaf : Field, index : Field, hash_path: [Field]) -> Field {
    let n = crate::array::len(hash_path);
    let index_bits = crate::to_bits(index, n as u32);
    let mut current = leaf;
    for i in 0..n {
        let path_bit = index_bits[i] as bool;
        let (hash_left, hash_right) = if path_bit {
            (hash_path[i], current)
         } else {
            (current, hash_path[i])
         };
```

Noir

```
pub enum Gate {
    Arithmetic(Expression),
    Range(Witness, u32),
    And(AndGate),
    Xor(XorGate),
    GadgetCall(GadgetCall),
    Directive(Directive),
}
```

```
pub enum OPCODE {
    #[allow(clippy::upper_case_acronyms)]
    AES,
    SHA256,
    Blake2s,
    MerkleMembership,
    SchnorrVerify,
    Pedersen,
    HashToField,
    EcdsaSecp256k1,
    FixedBaseScalarMul,
    ToBits,
}
```

pilcom and pil-stark

```
namespace Compressor(%N);
    pol constant S[12];
    pol constant Qm, Ql, Qr, Qo, Qk, QMDS, QCMul;
    pol commit a[12];
```

```
// Normal plonk ecuations
    pol a01 = a[0]*a[1];
    Qm*a01 + Ql*a[0] + Qr*a[1] + Qo*a[2] + Qk = 0;
```

```
{a[0], a[1], a[2], a[3], a[4], a[5], a[6], a[7], a[8], a[9], a[10], a[11]} connect
    {S[0], S[1], S[2], S[3], S[4], S[5], S[6], S[7], S[8], S[9], S[10], S[11]};
```

openzl

```
impl<S> poseidon::Specification<Compiler<S>> for S
where
    S: Specification,
{
    type Field = FpVar<S::Field>;

    #[inline]
    fn add(lhs: &Self::Field, rhs: &Self::Field, _: &mut Compiler<S>) -> Self::Field {
        lhs + rhs
    }

    #[inline]
    fn add_const(
        lhs: &Self::Field,
        rhs: &Self::ParameterField,
        _: &mut Compiler<S>,
    ) -> Self::Field {
        lhs + FpVar::Constant(rhs.0)
    }
```

Halo 2

```
trait NumericInstructions<F: FieldExt>: Chip<F> {
    /// Variable representing a number.
    type Num;

    /// Loads a number into the circuit as a private input.
    fn load_private(&self, layouter: impl Layouter<F>, a: Value<I

    /// Loads a number into the circuit as a fixed constant.
    fn load_constant(&self, layouter: impl Layouter<F>, constant

    /// Returns `c = a * b`.
    fn mul(
        &self,
        layouter: impl Layouter<F>,
        a: Self::Num,
        b: Self::Num,
    ) -> Result<Self::Num, Error>;
```

Halo 2

```rust
fn mul(
    &self,
    mut layouter: impl Layouter<F>,
    a: Self::Num,
    b: Self::Num,
) -> Result<Self::Num, Error> {
    let config = self.config();

    layouter.assign_region(
        || "mul",
        |mut region: Region<'_, F>| {
            // We only want to use a single
            // so we enable it at region of
            // cells at offsets 0 and 1.
```

Plonky2

```
let config = CircuitConfig::standard_recursion_config();
let mut builder = CircuitBuilder::<F, D>::new(config);

// The arithmetic circuit.
let initial = builder.add_virtual_target();
let mut cur_target = initial;
for i in 2..101 {
    let i_target = builder.constant(F::from_canonical_u32(i));
    cur_target = builder.mul(cur_target, i_target);
}

// Public inputs are the initial value (provided below) and the resu
builder.register_public_input(initial);
builder.register_public_input(cur_target);
```

Barretenberg

```
round_quad.q_x_1 = ladder[i + 1].q_x_1;
round_quad.q_x_2 = ladder[i + 1].q_x_2;
round_quad.q_y_1 = ladder[i + 1].q_y_1;
round_quad.q_y_2 = ladder[i + 1].q_y_2;

if (i > 0) {
    ctx->create_fixed_group_add_gate(round_quad);
```

ZK Garage

```
fn gadget(
    &mut self,
    composer: &mut StandardComposer<F, P>,
) -> Result<(), Error> {
    let a = composer.add_input(self.a);
    let b = composer.add_input(self.b);
    let zero = composer.zero_var();

    // Make first constraint a + b = c (as public input)
    composer.arithmetic_gate(|gate| {
        gate.witness(a, b, Some(zero))
            .add(F::one(), F::one())
            .pi(-self.c)
    });

    // Check that a and b are in range
    composer.range_gate(a, 6);
    composer.range_gate(b, 4);
```

Jellyfish

```rust
// Step 1:
// We instantiate a turbo plonk circuit.
//
// Here we only need turbo plonk since we are not using plookups.
let mut circuit = PlonkCircuit::<EmbedCurve::BaseField>::new_turbo_plonk();

// Step 2:
// now we create variables for each input to the circuit.

// First variable is x which is an field element over P::ScalarField.
// We will need to lift it to P::BaseField.
let x_fq = fr_to_fq::<_, EmbedCurve>(&x);
let x_var = circuit.create_variable(x_fq)?;

// The next variable is a public constant: generator `G`.
// We need to convert the point to Jellyfish's own `Point` struct.
let G_jf: Point<EmbedCurve::BaseField> = G.into();
let G_var = circuit.create_constant_point_variable(G_jf)?;

// The last variable is a public variable `X`.
let X_jf: Point<EmbedCurve::BaseField> = X.into();
let X_var = circuit.create_public_point_variable(X_jf)?;

// Step 3:
// Connect the wires.
let X_var_computed = circuit.variable_base_scalar_mul::<EmbedCurve>(x_var, &G_var)?;
circuit.enforce_point_equal(&X_var_computed, &X_var)?;

// Sanity check: the circuit must be satisfied.
```

PIC

```
let pi = VirtualQuery::new(0, Rotation::curr(), OracleType::Instance);

let pow_7 = |expr: VirtualExpression<F>| -> VirtualExpression<F> {
    let expr_squared = expr.clone() * expr.clone();
    let expr_pow_4 = expr_squared.clone() * expr_squared.clone();

    expr_pow_4 * expr_squared * expr
};
```

What should I choose??

- General cryptography
- Proving system developers
- Circuit developers
- End-users

A way forward?

- Common IOP
- Proof system composition
- Efficient black box, type safe, no magic

hello@geometry.xyz

@__geometry__

LONDON ST. HELIER BELGRADE TEL AVIV SINGAPORE