Succinct proofs of arbitrary computations

Balázs Kőműves
Faulhorn Labs & Codex storage
bkomuves@gmail.com

ELTE, Budapest
2024.09.30.

Introduction

Suppose I want to convince you that I know the discrete logarithm $d \in \mathbb{N}$ of an elliptic curve point $h \in \mathcal{C}$ in a large elliptic curve[1] $\mathcal{C}$, *without telling you* d. That is, $g^d = h$ where $g \in \mathcal{C}$ is a fixed generator. I can do this for example[2] using the *elliptic curve digital signature algorithm* (ECDSA): You produce a random challenge string, and I sign it with my private key d.

However, the ECDSA algorithm is based on special algebraic properties of elliptic curves. What if I want to convince you of something else? For example that I know a preimage of a given SHA256 hash. Or that I know a solution for Sudoku puzzle.

It turns out that it's possible to do so for any NP problem! Such a thing is usually called[3] a "zero-knowledge proof".

[1] over a finite field $\mathbb{F}$

[2] another option is Schnorr's protocol

[3] with some abuse of technical nomenclature...

Problem statement

Given a (deterministic) computer program $\mathbf{prg} : X \to Y$, we want to convince another party that $\mathbf{prg}(x) = y$ for some given $x \in X$ and $y \in Y$.

Of course we could just give them the program and let them run it. However, we can do better than that!

It's actually possible to convince a third party about such a claim in a very small (even constant!) space and time.

Think somewhere between a few hundred bytes and a few hundred kilobytes of proof size, and tens to hundreds of milliseconds of verification time![4]

[4] creating the proofs is expensive though

Nondeterministic advice

A slight but useful generalization of the above setting is when we also allow an extra, “secret” input $w \in W$ to the program:

$$\mathbf{prg} : X \times W \to Y$$

As an example, we can have a program which checks whether the solution of a Sudoku puzzle is valid. In this case $x \in X$ would be the puzzle, $w \in W$ would be the solution, and $Y = \texttt{bool}$. We can then convince somebody that we know a solution without actually telling them the solution! This is usually called the “zero-knowledge” property[5].

$w \in W$ is usually called “nondeterminstic advice” or sometimes “witness”.

[5] technically zero-knowledge means that the verifier learns *nothing* about w except that $\mathbf{prg}(x, w) = y$

Proofs vs. arguments

While people usually refer to these techniques as “zero-knowledge proofs”, these are actually not traditional mathematical proofs, but *computationally binding arguments*.

This means that a computationally bound[6] attacker is not able to create a false “proof” (except with a negligibly small probability).

However for brevity we often use the word “proof” anyway.

As we can make the amount of computation required for cheating arbitrarily large, and the probabilities arbitrarily small, in practice this is almost as good as a real mathematical proof.

[6] think needing something like $2^{120} \approx 10^{36}$ computational steps to be able to create a fake argument

Some possible applications

There are many possible applications of such a technology, for example:

- anonymous online identity
- privacy respecting age verification
- selective reveal of information (for example in healthcare or court)
- online voting
- scaling up blockchains
- privacy on public blockchains
- checking authenticity of news photographs
- certified binaries[7]
- etc etc

[7] not currently practical

But how is this possible?!

There are many different approaches, but essentially all of them boil down to the following:

- first we translate our computer program into a *set of mathematical equations* (involving polynomials over finite fields), such that the only solutions of these equations are the valid *execution traces* of our program
- then fix the variables corresponding to the input and output x and y, so there is only 1 solution remaining
- finally use some kind of mathematical trickery to prove that we know a solution of these equations

Since there is also at most one solution, and that has to correspond to the execution trace of our program, this "proves" that indeed $\mathbf{prg}(x) = y$.

What is a computation?

There are many different ways to represent computations. For this talk, we will use a very simple virtual CPU. While for example λ-calculus or combinator calculus are very nice abstractions, for our purposes a state machine (which a CPU is) is a better fit.

The reason for this is that we want to encode everything into polynomials, and it's complicated to put a tree-like structure into a polynomial, while the execution trace of a state machine is just a linear sequence.

So we will use a pretty traditional Harvard architecture, with only a few registers, and a very small instruction set. A difference from traditional computers is that the values in the memory and registers won't be machine words but finite field elements[8].

[8] you could emulate machine words but it's more expensive and a bit more complicated

VM design

We try to design a VM which is simple to implement. We will have:

- separate program storage (the reason for this is to simplify instruction decoding);
- traditional R/W memory, initialized with the input and also all the constants the program needs (to simplify the instruction set);
- at most 1 memory access per instruction;
- a traditional stack (cohabited with the memory);
- a few registers: the program counter (PC), the stack pointer (SP), an accumulator[9] (A), and maybe also an index register (X). We can use the memory for everything else.
- a very minimal instruction set;
- some addressing modes: for example absolute and relative to SP.

[9] The role of A is twofold: to ensure that there is only one memory access per instruction; and to a lesser extent, to ensure that there is only one argument per instruction.

Part I. - From programs to equations

The execution trace

When we execute a program in this VM, at every "processor cycle", we can write down the whole state of the CPU as a row vector. So after N cycles we will have an $N \times K$ matrix, called the *execution trace*.

The columns of this trace could for example consist of:

- one column for the time step (cycle count);
- a few columns for the current instruction to execute;
- columns for the values of each register (PC, SP, A...);
- a column for the zero flag Z;
- three columns to handle the memory (more about this later);
- more columns depending on various features we support.

We then want to prove that each step, the CPU state transition is valid.

Instruction encoding

For each possible machine instruction OP, we will need a *selector column* S_{OP} which is 1 at the cycles we execute that particular instruction, and 0 otherwise. There are a few different ways to achieve this:

- simply encode the program with as many such columns as many instructions we have. This is "low degree" but a lots of columns.
- encode instructions by a number I_{OP}, and use Lagrange polynomials for the selectors. This is a single column but "high degree":

$$S_{\mathsf{ADD}}(op) := \frac{(op - I_{\mathsf{MUL}})(op - I_{\mathsf{INC}})(op - I_{\mathsf{JMP}})\dots}{(I_{\mathsf{ADD}} - I_{\mathsf{MUL}})(I_{\mathsf{ADD}} - I_{\mathsf{INC}})(I_{\mathsf{ADD}} - I_{\mathsf{JMP}})\dots}$$

- encode instructions with columns b_i for each bit of the opcode. This is a good compromise between size and degree. For example if the opcode is $13_{10} = 1101_2$, we can have

$$S_{13} := b_3 \cdot b_2 \cdot (1 - b_1) \cdot b_0$$

Instruction lookup

Consider the following simple program (left) and its execution (right):

addr	instr	
0	NOP	
1	ADD	#5
2	MUL	#2
3	SUB	#3
4	JMP	2
5	...	

T	PC	instr		A'
0	0	NOP		0
1	1	ADD	#5	5
2	2	MUL	#2	10
3	3	SUB	#3	7
4	4	JMP	2	7
5	2	MUL	#2	14
6	3	SUB	#3	11
7	4	JMP	2	11
8	2	MUL	#2	22
9	3	...		...

We have to ensure that the instructions in the execution trace matches those in the program indexed by the program counter (PC) register.

We can use a so called *lookup argument* to ensure this. This will essentially prove that the set of $(\mathrm{PC}, instr)$ pairs on the right is a subset of the set of $(addr, instr)$ pairs on the left. More about this later.

State transition constraints

Consider a toy machine with a single register X and only 4 instructions: ADD, MUL, SET and JMP. So the execution trace will have 5 columns: T (cycle count), PC, the register X, the opcode to execute (OP), and its argument (ARG). We can additionally have the bits of the opcode b_0, b_1. We can now write down equations for the state transition function:

$$\begin{aligned} T' &= T + 1 \\ \mathsf{PC}' &= (1 - S_{\mathsf{JMP}})(\mathsf{PC} + 1) + S_{\mathsf{JMP}} \cdot \mathsf{ARG} \\ X' &= S_{\mathsf{ADD}}(X + \mathsf{ARG}) + S_{\mathsf{MUL}}(X \cdot \mathsf{ARG}) + S_{\mathsf{SET}} \cdot \mathsf{ARG} + S_{\mathsf{JMP}} \cdot X \end{aligned}$$

We also need the instruction selectors; these will be "advice columns":

$$\begin{aligned} \mathsf{OP} &= 2b_1 + b_0 \\ b_i(1 - b_i) &= 0 \\ S_{\mathsf{ADD}} &= (1 - b_1)(1 - b_0) \\ S_{\mathsf{MUL}} &= (1 - b_1)b_0 \\ &\dots \end{aligned}$$

Conditionals

Now let's try to add conditionals, for example BZ (branch if zero). For this, we first need a "zero flag" column Z, which is 1 when $X = 0$ and 0 otherwise.

We can do this with following trick. First define a new advice column $Y = X^{-1}$. Then the following equations ensure the above property:

$$Z = 1 - X \cdot Y$$

$$Z \cdot X = 0$$

After this, it's very easy to add the new BZ instruction:

$$\begin{aligned} \mathsf{PC}' = (1 - S_{\mathsf{JMP}} - S_{\mathsf{BZ}})(\mathsf{PC} + 1) + S_{\mathsf{JMP}} \cdot \mathsf{ARG} + \\ + S_{\mathsf{BZ}} \cdot \big\{ Z \cdot \mathsf{ARG} + (1 - Z)(\mathsf{PC} + 1) \big\} \end{aligned}$$

Stopping program execution

At some point we need to halt program execution, otherwise the trace would just grow indefinitely. We can use a dedicated HALT instruction for this. Also for technical reasons, the trace usually have to have a power-of-two size, and is circular: after the last row, it starts again at the first row.

To handle these issues, we can make the HALT instruction to behave like an infinite loop. We also need to “disconnect” the transition between the last row and the first row. We can do that for example by multiplying all equations by $(1-\mathcal{L}_{N-1})$ where $\mathcal{L}_k$ is the Lagrange polynomial taking 1 at row k and 0 in all the other rows:

$$0 = (1-\mathcal{L}_{N-1})\cdot\Big\{\mathsf{PC}' - \big[S_{\mathsf{JMP}}\cdot\mathsf{ARG} + (1-S_{\mathsf{JMP}})(\mathsf{PC}+1)\big]\Big\}$$

Memory, page I.

How to handle memory? A naive approach would be to add a column for every memory cell. But that's a really bad idea: We would need a *huuuuuge* amount of columns!

Instead, we will only write down the memory accesses in a table:

T	instr	A	A'
10	STA [5]	13	
13	LDA [17]		4
15	LDA [5]		13
18	LDA [29]		6
21	STA [17]	10	
22	LDA [29]		13
24	STA [5]	8	
27	LDA [5]		8
30	STA [29]	1	
31	LDA [17]		10

$\Rightarrow$

T	addr	R/W	value
10	5	W	13
13	17	R	4
15	5	R	13
18	29	R	6
21	17	W	10
22	29	R	13
24	5	W	8
27	5	R	8
30	29	W	1
31	17	R	10

Memory, page II.

But now we have to prove the consistency of that table! To do that, reorder first by address, and within that by time:

T	addr	R/W	value
10	5	W	13
13	17	R	4
15	5	R	13
18	29	R	6
21	17	W	10
22	29	R	6
24	5	W	8
27	5	R	8
30	29	W	1
31	17	R	10

$\rightsquigarrow$

$addr_*$	T_*	RW_*	val_*
5	10	W	13
5	15	R	13
5	24	W	8
5	27	R	8
17	13	R	4
17	21	W	10
17	31	R	10
29	18	R	6
29	22	R	6
29	30	W	1

Memory, page III.

Now we can start to write down some constraints:

- the reordered memory table is a permutation of the memory part of the execution trace
- address is monotonically increasing
- values do not change when the address is constant and memory access is "read"
- time is increasing when the address does not change

To be able to convert these into equations, we will need a so-called "permutation argument" (similar to the lookup arguments we needed for instruction lookup), and also inequalities. The latter is tricky, as finite fields do not have any natural ordering...

Memory equations

We can for example have two instructions: LDA and STA, loading into register A from memory and storing A into memory. Then our equations could look something like this:

$$
\begin{aligned}
0 &\le addr'_* - addr_* \\
0 &= \left[1 - (T'_* \overset{?}{>} T_*)\right] \cdot (addr'_* \overset{?}{=} addr_*) \\
0 &= (val'_* - val_*) \cdot (addr'_* \overset{?}{=} addr_*) \cdot (RW_* \overset{?}{=} \mathsf{R}) \\
A' &= S_{\mathsf{LDA}} \cdot val + S_{\mathsf{ADD}}(A + \mathsf{ARG}) + \dots \\
0 &= S_{\mathsf{STA}} \cdot (val - A) \\
addr &= \mathsf{ARG} + \mathsf{REL} \cdot \mathsf{SP} \\
RW &= (1 - S_{\mathsf{STA}}) \cdot \mathsf{R} + S_{\mathsf{STA}} \cdot \mathsf{W}
\end{aligned}
$$

We can create columns of the form $X \overset{?}{=} Y$ (meaning 1 if $X_i = Y_i$ and 0 otherwise) using the same trick we used to create the zero flag column.

Inequalities

Inequalities are tricky, because finite fields have no (natural) orderings. What we can do instead is to interpret $x \leq y$ as $x, y, y - x \in [0, N)$ where $N < |\mathbb{F}|/2$. This is called a *range check*. In practice we usually have $N = 2^n$.

The simplest way to prove this is to decompose the difference $d := y - x$ into bits:

$$\begin{aligned} d &= 2^{n-1}b_{n-1} + 2^{n-2}b_{n-2} + \cdots + 4b_2 + 2b_1 + b_0 \\ 0 &= b_i(1 - b_i) \qquad\qquad \text{for all } 0 \leq i < n \end{aligned}$$

However, for say a 32 bit range check this requires 32 columns...

A potentially cheaper apprach is to decompose into bytes or 16-bit words, and prove those smaller ranges via *lookup tables*.

Initializing memory

We want (some parts of) the memory initialized to given values when the VM execution starts. We definitely need this for the input, or we could also put constants in the memory instead of the program code (simplifying the ISA). If we use a von Neumann architecture, we would load the program code in memory too.

This looks quite tricky at first slight: We have to ensure that the very first read at given memory locations give certain values. Logically, we encode this as:

$$addr_* \neq addr'_* \quad \wedge \quad RW'_* = \mathsf{R} \quad \wedge \quad (addr'_*, val'_*) \in M$$

where M is table of $(addr, value)$ pairs for intializing the memory.

We can translate these into a (conditional) lookup argument, similar to what we we used for lookup up instructions for example.

Stack

We can simply include a stack the same way your CPU does: just reusing the existing memory and indexing it with a stack pointer.

However, dedicated stack machines also exist, for example Miden VM. In Miden the instructions pop their arguments from the top of the stack, and push the result back to stack. The top few elements of the stack are directly accessible (like registers, they are individual columns); the rest is not directly accessible, but "accumulated" in some form of "overflow table". An advantage of this approach is simpler instructions (no need for arguments).

For the details of how this works, see:

`https://0xpolygonmiden.github.io/miden-vm/design/stack/main.html`

Bootloading by hash

In some systems, especially where a von Neumann architecture is used, specifying the program (as part of the public input) can be expensive.

A trick which is often used to compress it (but also large inputs), is to put these expensive public inputs inside the nondeterministic advice (or private input) part instead, and put only their *cryptographic hashes* (which are very small) in the actual (expensive) public input part.

Of course you then have to *prove* that those hashes are really the hashes of the now private data; this increases the prover time instead, so this is always a tradeoff. Because traditional cryptographic hash functions like for example SHA256 are typically very expensive inside zero-knowledge proofs, new, “ZK-friendly” arithmetic hash functions were designed for these kind of applications.

Accelerator gadgets

There are some operations which are very expensive, but can be made significantly cheaper by dedicated "circuits". Two examples of this are range checks and hashing. This is very similar to physical CPUs, which also has dedicated circuitry for various specific tasks, including for example SHA256 hashing.

The idea is to build a completely separate proof only for a particular operation, and then connect it with the main "CPU" somehow, for example with lookup arguments (the dedicated circuit proves a table $(input, output)$ pairs, and then the main CPU can simply look up the result).

More examples of such "gadgets" are: range checks, 32-bit arithmetic operations, bitwise operations, hashing, elliptic curve operations, foreign field arithmetic, etc.

Part II. - The underlying math

How to prove the equations?

So far we have seen how to translate our virtual CPU into a set of equations. But how to prove that these equations hold?

First of all, we will encode the columns as values of *polynomials* over finite fields. Let $H = \{1, \omega, \omega^2, \ldots, \omega^{N-1}\} \subset \mathbb{F}$ a multiplicative subgroup of our field $\mathbb{F}$; then to a column vector A we will associate an *interpolation polynomial* $P_A(x)$ (of degree $N-1$) such that

$$P_A(\omega^k) = A_k$$

Now if an equation like for example $A_k(1 - A_k) = 0$ holds for all k, that equivalent to say that the derived polynomial $P_A(x)[1 - P_A(x)]$ has roots at $\underline{\text{all } \omega^k \in H}$, which is further equivalent with $Z(x) := \prod_k (x - \omega^k)$ dividing it. Then we can form the *quotient polynomial*

$$Q(x) := \frac{P_A(x) \cdot \big[1 - P_A(x)\big]}{Z(x)}$$

and our final equation will be $P_A(x) \cdot [1 - P_A(x)] = Q(x) \cdot Z(x)$, $\underline{\forall x \in \mathbb{F}}$.

The Schwartz-Zippel lemma

So we managed to reduce an equation which holds over a *subgroup* $H \subset \mathbb{F}$ to another one which holds *for all* $x \in \mathbb{F}$. How to prove such equations?

Reordering, we want to prove that $P_A \cdot [1 - P_A] - Q \cdot Z \equiv 0$. The left-hand side has degree at most $2(N-1)$, so, unless it's constant zero, it can have at most that many roots. Thus, if we select a random $\zeta \in \mathbb{F}$ and check the equation there (by substituting $x \mapsto \zeta$), with probability

$$1 - \frac{\text{deg}}{|\mathbb{F}|} \;=\; 1 - \frac{2(N-1)}{|\mathbb{F}|} \;>\; 1 - \epsilon$$

we will get a nonzero result, disproving the equation. This fact (and its generalizations) is called the "Schwartz-Zippel lemma".

In practice we often have $|\mathbb{F}| \gg \text{deg}$, for example $|\mathbb{F}|$ can be around 2^{256}, while deg $< 2^{30}$. In the case $\mathbb{F}$ is too small, we either repeat the randomized test several times, or use a *field extension* $\mathbb{F} \subset \widetilde{\mathbb{F}}$ which is big enough.

Interactive Oracle Proofs (IOPs)

IOPs are a mathematical abstraction which is convenient to work with, and prove propeties about.

An IOP is an interactive protocol between a prover and a verifier, in which the prover can also send magic black boxes, called "oracles" to the verifier, who can ask questions from the boxes. In our case, these magic boxes contain (huge) polynomials, and the verifier can query evaluations at discrete points (and get an answer instantly).

This is a nice level of abstraction to describe proof systems. Of course such magic black boxes do not exists in real life, however, we have something close enough to make it work.

Polynomial commitments

Polynomial commitment schemes are a very important cryptographic building block of ZK proofs. They allow to first *commit* to polynomials, by producing a short commitment (this is like a hash or fingerprint: If you change the polynomial, the commitment also changes, except with an extremely low probability), and then the prover can prove the *evaluations* these polynomials at different locations:

$$\begin{aligned} \texttt{commit} : \mathbb{F}[x] &\to \mathcal{G} \\ P &\mapsto \widehat{P} \end{aligned}$$

$$\begin{aligned} \texttt{eval} : \mathbb{F}[x] \times \mathbb{F} &\to \mathbb{F} \times \mathcal{H} \\ \big(P\ ,\ \zeta\ \big) &\mapsto \big(P(\zeta)\ ,\ \pi\big) \end{aligned}$$

$$\begin{aligned} \texttt{verify} : G \times H \times \mathbb{F} \times \mathbb{F} &\to \texttt{bool} \\ \big(\widehat{P}\ ,\ \pi\ ,\ \zeta\ ,\ P(\zeta)\big) &\mapsto \texttt{true} \end{aligned}$$

So these can replace the oracles in IOPs. Two popular such schemes are KZG and FRI commitments.

Example proof flow

So let's say we have an equation we want to prove, something like

$$P(x) := S_1(x)\big[A(\omega\cdot x) - A(x)^2 - B(x)\big] + S_2(x)\big[A(\omega\cdot x) - C(x)^3\big] = 0 \quad \underline{\forall x \in H}$$

- first we *commit* to the polynomials A, B, C. Some polynomials, for example S_1 and S_2 can be also *pre-committed* public information
- next we compute the quotient polynomial $Q(x) := P(x)/Z_H(x)$, and we commit to $Q(x)$ too
- then we choose a uniformly random $\zeta \in \mathbb{F}$. In the interactive setting, this can be chosen by the verifier; in the non-interactive setting, it can be chosen using the Fiat-Shamir heuristic, by hashing all the above commitments
- the verifer can now ask for the evaluations (and corresponding proofs) of $A(\zeta)$, $A(\omega\zeta)$, $B(\zeta)$, $C(\zeta)$, $S_1(\zeta)$, $S_2(\zeta)$, $Q(\zeta)$
- it can then check the evaluation proofs, and finally check the equation:

$$Q(\zeta)\cdot Z_H(\zeta) = S_1(\zeta)\big[A(\omega\cdot\zeta) - A(\zeta)^2 - B(\zeta)\big] + S_2(\zeta)\big[A(\omega\cdot\zeta) - C(\zeta)^3\big]$$

Permutation arguments, page I.

To handle memory, we needed a component which can prove that the values of two column vectors (represented by values of two polynomials P, Q on a subgroup H) are the permutation of each other (also called a “multiset equality check”).

In fact, we needed this for a pair of *tuples of columns*; however, there is a standard trick to “compress” several columns (polynomials) into one by taking a *random linear combination*: Choose a random $\beta \in \mathbb{F}$, and simply combine with powers of β:

$$\widetilde{P}(x) \;:=\; \sum_{i} \beta^{i} \cdot P_{i}(x).$$

If $|\mathbb{F}|$ is big enough, there is only a negligibly small chance of cancellations ruining our “proof”.

So we will only consider the case of two columns (values of two polynomials) being permutations of each other.

Permutation arguments, page II.

So we need to prove that $\{P(h) : h \in H\} = \{Q(h) : h \in H\}$. The underlying idea is simple: Let's form two new polynomials

$$A(x) := \prod_{h \in H} \big(x - P(h)\big) \qquad \text{and} \qquad B(x) := \prod_{h \in H} \big(x - Q(h)\big)$$

and then prove that $A \equiv B$ by the usual way: Choosing a random $\eta \in \mathbb{F}$ and checking that $A(\eta) = B(\eta)$.

An alternative, more efficient version would be to form

$$C(x) := \frac{A(x)}{B(x)} = \prod_{h \in H} \frac{x - P(h)}{x - Q(h)}$$

and prove that $C \equiv 1$ the same way.

But how to convince the verifier that C (or A and B) were indeed formed correctly? For that, we can use a so-called *grand product argument*.

Grand product argument

So let's assume we computed (and committed to)

$$C(x) := \frac{A(x)}{B(x)} = \prod_{h\in H} \frac{x - P(h)}{x - Q(h)}$$

and now we have to convince the verifier that we haven't cheated: C is indeed what we claim it to be, and $C(\eta) = 1$ for a random chosen $\eta \in \mathbb{F}$.

To do this, form the vector of *partial products*

$$U_k := \prod_{i=0}^{k-1} \frac{\eta - P(\omega^i)}{\eta - Q(\omega^i)} \in \mathbb{F}$$

and encode it as usual into a polynomial $\mathcal{U}$ such that $\mathcal{U}(\omega^k) = U_k$. We can now write down some equations: $\forall x \in H = \{\omega^i \,:\, 0 \leq i < N\}$

$$\begin{aligned} 0 &= \mathcal{L}_0(x)(\mathcal{U}(x) - 1) & \iff \mathcal{U}(1) = 1 \\ \frac{\mathcal{U}(\omega x)}{\mathcal{U}(x)} &= \frac{\eta - Q(x)}{\eta - P(x)} & \end{aligned}$$

These ensure that $U_N = 1$ (this works out because $\omega^N = \omega^0 = 1$).

Lookup arguments

With lookup arguments, we want to prove that the set of values of a column is a subset of another column; or when encoded as polynomials:

$$\mathcal{P} := \left\{ P(\omega^i) \;:\; 0 \le i < N \right\} \;\subset\; \left\{ T(\omega^k) \;:\; 0 \le k < N \right\} =: \mathcal{T}$$

(as before, we can compress tuples of columns into single columns).

There are quite a few different approaches; a relatively simple (and also the first) one is called **"plookup"**. This relies on the following trick: Fix an (arbitrary) ordering on $\mathbb{F}$, and sort both $\mathcal{T}$ and $\mathcal{S} := \mathcal{P} \cup \mathcal{T}$ (as multisets). Then

$$\mathcal{P} \subset \mathcal{T} \quad\Longleftrightarrow\quad \left\{ (\mathcal{S}_k, \mathcal{S}_{k+1}) \right\} = \left\{ (\mathcal{T}_j, \mathcal{T}_{j+1}) \right\} \cup \left\{ (\mathcal{P}_i, \mathcal{P}_i) \right\}$$

This we can prove with a permutation (or multiset equality) argument, which we already seen how to do.

The linear combination trick

When you have a large number of equations, proving them individually can become quite expensive. However, here again one can use the *linear combination trick*: Just choose some random coefficients c_i, and combine them into a single equation:

$$\forall x \in H.\ \sum c_i \cdot P_i(x) = 0 \quad \text{``}\Longleftrightarrow\text{''} \quad \forall i.\ \forall x \in H.\ P_i(x) = 0$$

In practice you can even choose $c_i = \alpha^i$ (of course all this assumes that $|\mathbb{F}|$ is big, let's say $|\mathbb{F}| > 2^{120}$).

This is especially useful when $H \subset \mathbb{F}^\times$ is a multiplicative subgroup, because quotienting by $Z_H(x)$ is a linear operation, so you can combine first and quotient only for the combined equation. As computing the quotient is also expensive, this is a big win.

Implementations

Modular IOPs

We have seen that already the individual components of a zkVM can be quite involved (conside the memory for example), and combining them into a full zkVM can be very tedious and error-prone. It's also very unlikely to get the design right at the first try, so this process needs to be repeated many times.

Clearly writing down the equations (and implementing them in code) manually is not the right approach. Instead we should have a DSL allowing to describe complex IOPs in a modular way. At the same time we also need to compute the solutions of those equations.

Furthermore, applying Fiat-Shamir or adding blinding factors is also error-prone; that should be handled by a compilation process too.

I believe this is a very important problem to solve; I have yet to see anything that's even a step in the right direction. I'm guessing that all zkVM projects have some internal tooling, but I would bet that they are all unsatisfactory...

The big picture

These are the moving parts of a zkVM, from the higher levels of abstract towards lower levels:

- compiler from a high-level language to VM instructions
- a traditional VM implementation executing VM code
- the combined IOP proving the VM execution
- IOP building blocks like memory, lookups, range checks, etc
- converting the IOP into a proof system, by replacing the oracles with polynomial commitments, and interaction by Fiat-Shamir (this can be done manually or via a compiler)
- optionally adding blinding factors for full zero-knowledge
- a polynomial commitment scheme implementation
- underlying algebra library (finite fields, elliptic curves, etc)

Some existing zkVM-s

zkVM-s like this are very popular right now, and there are many project implementing such VMs, for example:

- Cairo by StarkWare - the original one; custom ISA, “read-only” (or write-once) memory
- Risc0 - the first one targeting RISC-V architecture (via LLVM)
- Miden - stack machine, Merkle-ized AST, data bus
- Triton VM - similar design to Miden, but core features only
- SP1 - another RISC-V compatible one, performance oriented
- Valida - custom ISA with optimizing LLVM frontend
- Jolt - based purely on lookup tables
- Nexus - based on folding schemes, close to RISC-V
- Lurk - a LISP, so quite different from the others (which all follow the CPU approach)
- various zkEVM-s like Polygon, zkSync, Scroll, etc (targeting the Ethereum Virtual Machine)

Tutorial implementation

I hope to implement such an educational zkVM in a self-contained Haskell package.

But I haven't had time to work on it in the last 2-3 weeks, and it's a rather nontrivial amount of work...

Especially the modular construction of IOPs is not a solved problem (I'm not aware of any implementation of this idea).

That said, Haskell implementation coming "soon" (tm)![10]

[10] within the next 5 years or so! pinky promise!!!