This is part of the article “Compiling to NIVC ZKVMs”. Feedback is welcome

Motivation

An arithmetisation describes how computation is structured in order to make it easier to prove properties about that computation. While the simplest way of expressing a computation using fields is with arithmetic circuit, circuits are unstructured. This lack of structure makes it very difficult to implement SNARKS and motivates the quest for a suitable abstraction or arithmetisation.

When designing circuits, knowing which types of operations are required and how many times they are used in the circuit guides the design of the prover for most proving systems. In other words, the performance of a proof system may be constrained on the chosen arithmetisation. Some representations such as R1CS are more fixed, others like Plonkish are more granular. This granularity offers control and potential optimisation at the expense of overall complexity of the proving system as a whole.

[

Screenshot 2024-01-22 at 22.46.35

778×468 32.3 KB

](https://europe1.discourse-cdn.com/standard20/uploads/anoma1/original/1X/a619a6e895f5f67732b355f3cd1bf21a331d7584.png)

In contrast to a general purpose proving system, a ZKVM with a fixed set of instructions allows us to tune the proving system to a particular circuit or set of circuits. This restrictions opens the door for further optimisation, as we’ve seen with STARK ZKVMs, where each prover is tailored to a particular ZKVM.

Recursive schemes offer a much larger room for optimising their provers. In choosing an arithmetisation for a recursive scheme, several factors must be taken into account:

- Flexibility
- Complexity

Since we are designing a compiler to ZKVMs, this compiler must understand the constraints of the IR or arithmetisation in order to optimise the efficiency of the prover.

R1CS

\textit{R1CS}

uses constraints of the form Az \times Bz = Cz

where A

, B

, and C

are \textit{linear combinations}

of variables and constants encoded in witness z

.

Folding schemes such as Nova (and its successor SuperNova) only work over R1CS instances, while the HyperNova scheme is defined over CCS. As we’ll see, CCS is a generalisation of both R1CS and Plonkish. The folding scheme ProtoStar is itself a generalisation of Nova defined over CCS, and thus over Plonkish and R1CS.

Compared to other arithmetisations, the main property of R1CS is that it is simple. R1CS-based folding schemes such as Nova achieve most of the benefits of folding, but they can’t have gates. Custom gates are powerful but complex. However, lookups can arguably be used as an alternative to custom gates.

The simplicity of these folding schemes on top of R1CS derives from the simplicity of R1CS. From an engineering perspective, they are likely to be more efficient than Plonkish-based folding schemes on the get-go, in a naive implementation, but they have less room for optimisation.

AIR

Algebraic Intermediate Representation (AIR) is a particularly structured kind of Plonkish circuit, generally associated with STARKs.

AIR is depicted by an execution trace, i.e. a matrix of field elements where each column can be viewed as a register and each row as a moment in time. Thus the execution trace represents the value of registers over time. That a certain computation was performed correctly is equivalent to proving that the execution trace satisfies certain properties.

The main two constraints in AIR are transition constraints and boundary constraints. Transition constraints represent the evolution of the trace, that is, that a each time step, a certain relation holds between the current state and the next state. Boundary constraints assert that a register takes on a certain value at some point in time.

While AIR is the intermediate representation that STARKWARE uses, most SNARKs can be easily tweaked to support both Plonkish and AIR.

Plonkish

Plonkish is a generalisation of AIR that includes a new column type of selectors.

Both Plonkish and AIR are conceptualised over the idea of an execution trace. However, Plonkish circuits have selectors, i.e. columns of boolean values that allow the proving system to merge several constraints into a single one, while AIR arithmetisation doesn't. This makes Plonkish circuits more general.

A Plonkish circuit-satisfiability relation consists of three modular relations, namely:

- A high-degree gate

relation checking that each custom gate is satisfied

- A permutation

relation checking that different gate values are consistent if the same wire connects them

- A lookup

relation checking that a subset of gate values belongs to a pre-processed table.

Unlike R1CS, which has a fixed structure, Plonkish circuits are a family of circuits defined in terms of a rectangular matrix of values. We refer to rows, columns, and cells of this matrix with the conventional meanings.

Among other proving systems, Halo2 uses this arithmetisation, although it provides a higher-level API with abstractions such as gates and chips. A gate in Halo2 is a collection of linearly independent constraints which must all be satisfied in a given row. A chip may apply multiple gates to a region of the trace matrix, and the gates may have been defined with the assumption that they will always be applied in a certain order. This is achieved by using "rotation" of columns, in order to reference values from neighbouring rows. This level of abstraction unfortunately discards a lot of valuable information about the potential relationship between different gates.

ProtoStar was also designed with Plonk as a target backend. Due to their modular architecture, their folding scheme proved to be flexible enough to be implementable over CCS when it came out.

Formal description

Formally, a Plonkish structure S

, S_{Plonkish}=(m,n,l,t,q,d,e,g,T,s)

, consists of:

- a multivariate polynomial g

in t

variables where g

is expressed as a sum of q

monomials and each monomial has a total degree at most d

. That is \begin{equation} *g(X_1,...,X_t) = \sum_{i=0}^q k_i \cdot X_1^{i_1} ... X_t^{i_t} \end{equation*}

, where i_1 + ... + i_t \leq d

for all i

.

- a vector of constants called \textit{selectors}

s \in \mathbb{F}^e

- a set of m

constraints. Each constraint i

is specified via a vector T_i

of length t

, with entries in the range {0,...,n+e-1}

.

\begin{equation} T_i := [T_{i_0}, ..., T_{i_{t-1}}] \; \text{(constraint)} \end{equation}

That is, if j < n-l

, then T_i[j]

will be a private input w_j

. If j \geq n-l

and j < n

, then T_i[j]

will be a private input x_j

. Otherwise, T_i[j]

will be a selector.

- size bounds m, n, l, t, q, d, e \in \mathbb{N}
- m

is the number of constraints (i.e. rows in the Plonkish matrix representation)

- n

is the number of private and public inputs (i.e. columns in the Plonkish matrix representation, excluding selectors)

- l

is the number of public inputs (n-l

is the number of private inputs)

- t

is the number of variables in the polynomial g

- q

is the number of monomials that compose g

- d

is the maximum degree of g

- e

is the number of selectors

- m

is the number of constraints (i.e. rows in the Plonkish matrix representation)

- n

is the number of private and public inputs (i.e. columns in the Plonkish matrix representation, excluding selectors)

- l

is the number of public inputs (n-l

is the number of private inputs)

- t

is the number of variables in the polynomial g

- q

is the number of monomials that compose g

- d

is the maximum degree of g

- e

is the number of selectors

A Plonkish instance consists of public input x \in \mathbb{F}^l

. A Plonkish witness consists of a vector w \in \mathbb{F}^{n-l}

. A Plonkish structure-instance tuple (S, x)

is satisfied by a Plonkish witness w

if \begin{equation} g(z[T_i[1]], ..., z[T_i[t]]) = 0 \end{equation}

for all i \in { 0,..., m-1}

, where z = (w, x, s) \in \mathbb{F}^{n+e}

CCS

Customisable Constraint System (CCS) is generalisation of R1CS and Plonkish that enables the use of high-degree gates while not requiring permutation arguments. There are costless reductions from instances of R1CS and Plonkish instances to equivalent CCS instances.

There's been recently a steady effort of abstracting out components in proving systems and thus enabling the decoupling between arithmetisation and backend. CCS is a consequence of this efforts.

However, there is no practical implementation of the CCS arithmetisation. This renders folding schemes such as HyperNova only theoretical.

Formal description

Formally, A CCS instance consists of public input x \in \mathbb{F}^l

. A CCS witness consists of a vector w \in \mathbb{F}^{n-l-1}

. A CCS structure-instance tuple (S, x) is satisfied by a CCS witness w if

\begin{equation} \sum_{i=0}^{q-1} c_i \cdot \circ_{j \in S_i} M_j \cdot z = \mathbf{0} \end{equation}

, where z = (w, 1, x) \in \mathbb{F}^n

.

M_j \cdot z

denotes matrix-vector multiplication, \circ

denotes the Hadamard (entry-wise) product between vectors, and 0 is an m

-sized vector with entries equal to the the additive identity in \mathbb{F}

.

Expanded, this equation looks like:

\begin{equation*} c_0 \cdot \overbrace{(M_{j_{0}} \cdot z \circ ... \circ M_{j_{|{S_{0}}|-1}} \cdot z)}^{j_i \in S_0} + ... + c_{q-1} \cdot \overbrace{(M_{j_{0}} \cdot z \circ ... \circ M_{j_{|{S_{q-1}}|-1}} \cdot z)}^{j_i \in S_{q-1}} = \mathbf{0} \end{equation*}

A CCS structure S

consists of:

- a sequence of matrices M_0,..., M_{t-1} \in \mathbb{F}^{m\times n}

with at most N = \Omega(max(m, n))

non-zero entries in total (think of these matrices as a generalisation of the A,B,C

matrices in R1CS that encode the constraints).

- a sequence of q multisets [S_0,...,S_{q-1}]

, where an element in each multiset is from the domain {0,..., t-1}

and the cardinality of each multiset is at most d (think of S_i

as containing the pointers to the matrices M_i

for each of the addends in the CCS equation).

- a sequence of q constants [c_0,...,c_{q-1}]

, where each constant is from \mathbb{F}

- size bounds m, n, N, l, t, q, d \in N

where n \ge l

- m

is the number of constraints (i.e. rows in the Plonkish matrix representation)

- n

is the number of private and public inputs (i.e. columns in the Plonkish matrix representation, excluding selectors)

- N

is the total number of non-zero entries in M_0,...,M_{t-1}

- l

is the number of public inputs (thus n-l

is the number of private inputs)

- t

is the number of M

matrices. (e.g. in R1CS, which can be seen as an instance of CCS, there are 3 M

matrices: A

, B

and C

)

- q

is the number of addends in the CCS equation (i.e. q = |{S_i}|

)

- d

is the upper bound of the cardinality of each S_i

- m

is the number of constraints (i.e. rows in the Plonkish matrix representation)

- n

is the number of private and public inputs (i.e. columns in the Plonkish matrix representation, excluding selectors)

- N

is the total number of non-zero entries in M_0,...,M_{t-1}

- l

is the number of public inputs (thus n-l

is the number of private inputs)

- t

is the number of M

matrices. (e.g. in R1CS, which can be seen as an instance of CCS, there are 3 M

matrices: A

, B

and C

)

- q

is the number of addends in the CCS equation (i.e. q = |{S_i}|

)

- d

is the upper bound of the cardinality of each S_i

Representing R1CS in CCS

As an example, the R1CS equation, (A \cdot z) \circ (B \cdot z) - C \cdot z = 0

can be represented with CCS as S_{CCS}=(n,m,N,l,t,q,d,[M_0, M_1, M_2], [S_1, S_2], [c_1, c_2])

, where m, n, N, l

are from R1CS and t=3

, q=2

, d=2

, M_0=A

, M_1=B

, M_2=C

, S_1={0,1}

, S_2={2}

, c_0=1

, c_1 = -1

. The relation then becomes:

\begin{equation*} 1 \cdot \overbrace{(A \cdot z \circ B \cdot z)}^{S_0 = {0,1}} + (-1) \cdot \overbrace{(C \cdot z)}^{S_1={2}} = 0 \end{equation*}

Representing Plonkish in CCS

How does the structure of CCS compare to Plonkish?

S_{Plonkish}=(m,n,l,t,q,d,e,g,T,s)

S_{CCS}=(m,n,N,l,t,q,d,[M_0,...,M_{t-1}], [S_0,...,S_{q-1}], [c_0,...,c_{q-1}])

How do we derive M_0,..., M_{t-1}

in S_{CCS}

? Each polynomial constraint in S_{Plonkish}

corresponds to some row in M_i

, so, it suffices to specify how the ith row of these matrices is set.

For all j \in {0, 1, ... , t - 1}

, let k_j = T_i[j]

(recall that t

is the number of variables in g

and T_i

represents a constraint in vector form). So k_j

is one of the values in a constraint.

If k_j \geq n

(i.e. if k_j

is greater than the number of public and private inputs - it points to a selector), then we set M_j[i][0] = s[k_j - n]

. Otherwise, we set M_j[i][k_j] = 1

. We set S_{CCS}.N

to be the total number of non-zero entries in M_0, ... , M_{t-1}

.

Each c_i

in [c_0,...,c_{q-1}]

is the coefficient of the ith monomial of g

. For i \in {0, 1, ... , q - 1}

, if the ith monomial contains a variable j

, where j \in {0, 1, ... , t - 1}

, add j

to multiset S_i

with multiplicity equal to the degree of the variable.