polygon zkEVM

Technical Document

Polynomial Identity Language (PIL): A Machine Description Language for Verifiable Computation

v.1.0

February 13, 2023

Contents

1 The Language

1.1 Introduction

Polynomial Identity Language (PIL) is a novel domain-specific language useful for defining eAIR constraints. The aim of creating PIL is to provide developers with a holistic framework for both constructing programs through an easy-to-use interface and abstracting the complexity of the proving mechanisms.

One of the main peculiarities of PIL is its modularity, which allows programmers to define parametrizable programs, called `namespaces`, which can be instantiated from larger programs. Building programs in a modular manner makes it easier to test, review, audit and formally verify even large and complex programs. In this regard, by using PIL, developers can create their own custom namespaces or instantiate namespaces from some public library.

Many other domain-specific languages (DSL) or tool stacks, such as Circom or Halo2, focus on the abstraction of a particular computational model, such as an arithmetic circuit. However, recent proof systems such as STARKs have shown that arithmetic circuits might not be the best computational models in all use cases. Given a complete programming language, computing a valid proof for a circuit satisfiability problem, may result in long proving times due to the overhead of re-used logic. Opting for the deployment of programs, with their low-level programming, shorter proving times are attainable, especially with the advent of proof/verification-aiding languages such as PIL.

1.2 Creating a Simple Program

To describe the fundamentals of the language itself, let us create a simple PIL program that models the computation of the product of two integers. Consider a program that, at each step, takes two input numbers and multiplies them. Naturally, this program will be referred to as the Multiplier program. This program can be modeled using 3 polynomials: 2 referring to the inputs that are going to be multiplied $\texttt{freeIn}_1, \texttt{freeIn}_2$, and 1 referring to the output of the computation itself `out`. As it can be observed, the output column will exhibit a correct behavior if and only if the following identity is satisfied:

$$\texttt{out} = \texttt{freeIn}_1 \cdot \texttt{freeIn}_2.$$

A concrete example of a correct execution trace of the Multiplier program can be seen in Figure 1. The relation above is satisfied in each of the rows of the execution trace, which means that the output column is filled with correct values.

row	$\texttt{freeIn}_1$	$\texttt{freeIn}_2$	out
1	4	2	8
2	3	1	3
3	0	9	0
4	7	3	21
5	4	4	16
6	5	6	30
⋮	⋮	⋮	⋮

Figure 1: An example of a valid execution trace for the Multiplier program.

As it can be seen, there exists a noticeable difference between the behavior of the input columns and the output column which suggests the following classification,

- **Free Input Polynomials:** These are columns that are in charge of introducing the various inputs to the computation. They are referred to as “free" because at every clocking of the computation, their values do not strictly depend on any previous iteration. These are analogous to independent variables of the entire computation.

- **State Variables:** These are the columns that compose the state of the program. Here state refers to the set of values that represent the output of the program at each step and, if we are in the last step, the output of the entire computation.

We can now write the corresponding PIL program for the Multiplier program:

```
namespace Multiplier(2**10);

// Polynomials
pol commit freeIn1;
pol commit freeIn2;
pol commit out;

// Constraints
out = freeIn1*freeIn2;
```

The reserved keyword `namespace` is used to frame the scope of the program definition. Inside it, one should define the polynomials used by its program and the constraints among the defined polynomials. A namespace has to be instantiated with a unique name (Multiplier) together with an argument representing the length of the program, that is, the number of rows (in this case, 2^{10}) of any execution trace of that program. Besides, the `commit` keyword allows the compiler to identify the corresponding polynomial as **committed**. Committed polynomials are opposed to **constant** polynomials, which are polynomials that are not allowed to change among any execution of the same program. That is, constant polynomials are inherent to the computation itself. This is important from the proving perspective since constant polynomials are publicly known by all parties. However, this is not the case for committed polynomials, which are, in most cases, only known by a party. More on constant polynomials will be added below.

One should observe that, of course, the former design of the Multiplier program is not unique. This design is highly not scalable to more complex operations since the number of committed polynomials grows linearly with the number of operations we want to perform. For example, designing a program that computes the result of performing 2^{10} operations following the previous design would require 2^{10} committed polynomials, which is far from being practical.

However, following another design strategy can easily reduce the 2^{10} committed polynomials to a single one by the introduction of another polynomial that flags the starting row of each operation. Together with a third polynomial holding the result of the operation, only a total amount of 3 columns will be needed. Returning to the initial Multiplier program, the idea is to introduce a **constant** polynomial called `RESET` that will evaluate to 1 in odd rows and 0 otherwise (see Figure 2). Nonetheless, this design will also decrease the number of multiplications that can be checked given the same number of rows as the initial design. More concretely, using the initial design we can check one multiplication per row, meanwhile adopting this new strategy will half the number of possible multiplication.

row	freeIn	RESET	out
1	4	1	0
2	2	0	4
3	3	1	8
4	1	0	3
5	9	1	3
6	0	0	0
7	0	1	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$

Figure 2: An example of a valid execution trace for the new design of the `Multiplier` program.

Observe that, whenever `RESET` equals 1, the value of the `out` polynomial equals the result of multiplying the previous two values of the `freeIn` polynomial. In the intermediate steps (that is, when `RESET` is equal to 0), the `out` polynomial stores the first input of the multiplication.

Of course, we need to adapt the Multiplier constraint to reflect the correctness of the `out` polynomial to the new design. One can observe the following constraint:

$$\texttt{out}' = \texttt{RESET} \cdot \texttt{freeIn} + (1 - \texttt{RESET}) \cdot (\texttt{out} \cdot \texttt{freeIn})$$

completely describes the new design. In PIL, a tick ' (which is read "prime") over a polynomial is used to denote the value in the next row of such polynomial. In the case of polynomials defined over a multiplicative subgroup G of a prime field $\mathbb{F}$ with generator g, the prime notation is equivalent to the polynomial $\texttt{out}'(X) := \texttt{out}(Xg)$.

To see that the previous constraint completely describes our new Multiplier design, let us distinguish between two cases:

- When `RESET` is equal to 1, the above constraint becomes:

  $$\texttt{out}' = \texttt{freeIn}.$$

  Hence, we are setting the `freeIn` polynomial's value in the current row into the `out` polynomial's value of the next row.

- When `RESET` is equal to 0, the above constraint becomes:

  $$\texttt{out}' = \texttt{out} \cdot \texttt{freeIn}.$$

  Hence, this constraint is stating that the `out` polynomial's value in the next row will become the product of the value of `freeIn` in the last two rows, the more distance contained in the `out` polynomial (as in PIL we are not allowed to access to more than one previous row's values).

The optimized design of the Multiplier program can be written in PIL as follows:

```
namespace Multiplier(2**10);

// Constant Polynomials
pol constant RESET;

// Committed Polynomials
pol commit freeIn;
pol commit out;

// Constraints
out' = RESET*freeIn + (1-RESET)*(out*freeIn);
```

Observe that now, the polynomial `RESET` is defined with the reserved keyword `constant`, because it does not change among several executions of the same program. Finally, note that the same design can be extended for a much larger amount of multiplications without needing to modify the PIL itself. Instead, we simply would extend the `RESET` polynomial as follows:

$$\texttt{RESET} = \begin{cases} 1, & \text{if } i \equiv 0 \pmod{n} \\ 0, & \text{otherwise} \end{cases}$$

where i represents the row number (starting from 0) and n refers to the number of operations.

1.3 Compilation

The previous PIL program is almost ready to be compiled into a JSON file using the pilcom package [Her22a]. This file is a basic JSON representation of the PIL program (with some extra metadata) that will be consumed later on by the `pil-stark` package [Her22b] to generate a STARK proof. However, there is a strong restriction when dealing with PIL's constraints. Let S be the set of all polynomials defined over a field $\mathbb{F}$ appearing in the PIL program. Formally, a constraint is a polynomial identity $f = 0$ where $f \in \mathbb{F}[S, S']$ where S' is the set of all the shifted polynomials $p(gX)$ with $p \in S$. The restriction in PIL is the following: **the degree of f must be less or equal to** 2.

For example, recall the previous PIL program for the optimized Multiplier program. The constraint

$$\texttt{out}' = \texttt{RESET} \cdot \texttt{freeIn} + (1 - \texttt{RESET}) \cdot (\texttt{out} \cdot \texttt{freeIn}).$$

can be viewed as the polynomial identity $f = 0$ where f equals to

$$\texttt{out}' - \texttt{RESET} \cdot \texttt{freeIn} + (1 - \texttt{RESET}) \cdot (\texttt{out} \cdot \texttt{freeIn})$$

which belongs to $\mathbb{F}[\texttt{out}, \texttt{out}', \texttt{RESET}, \texttt{freeIn}]$ but **does not have degree less or equal than** 2, because it contains the monomial

$$\texttt{RESET} \cdot \texttt{out} \cdot \texttt{freeIn}.$$

The idea that PIL has introduced to solve this limitation is to create a new polynomial, conveniently named `carry`, that will store the product $\texttt{out} \cdot \texttt{freeIn}$, that is,

$$\texttt{carry} = \texttt{out} \cdot \texttt{freeIn}.$$

This kind of polynomial will be called **intermediate**. Observe that the prover does not need to provide intermediate polynomials since they can be derived directly from the committed and constant polynomials. Only the way of computing it is strictly necessary.

Now, the former PIL program can be modified to introduce this newly intermediate `carry` polynomial:

```
namespace Multiplier(2**10);

// Constant Polynomials
pol constant RESET;

// Committed Polynomials
pol commit freeIn;
pol commit out;

// Intermediate Polynomials
pol carry = out*freeIn;

// Constraints
out' = RESET*freeIn + (1-RESET)*carry;
```

At this point, given a (valid) PIL file (ending with the `.pil` extension) the pilcom compiler will return a JSON file. For example, the Multiplier program's PIL located in `multiplier.pil` can be compiled by invoking the following command:

```
pilcom$ node src/pil.js multiplier.pil -o multiplier.json
```

Apart from the JSON file, the compiler will throw a debugging message into the console:

```
Input Pol Commitments: 2
Q Pol Commitmets: 1
Constant Pols: 1
Im Pols: 1
plookupIdentities: 0
permutationIdentities: 0
connectionIdentities: 0
polIdentities: 1
```

In the previous message, it can be checked that the number of committed polynomials (`Input Pol Commitments`), the number of constant polynomials (`Constant Pols`), the number of intermediate polynomials (`Im Pols`) and the total number of identity constraints (`polIdentities`) agrees with the corresponding PIL code.

Since one of the key features of PIL is that it allows modularity, a dependency inclusion feature among different `.pil` files have been developed. To briefly show this feature, a configuration `.pil` file containing the degree bound for polynomials will be created and be used along several PIL programs to remove magic numbers like 2^{10}. More generically, `config.pil` will typically include some configuration-related components, shared among various programs. To declare a constant, it can do it using the `constant` keyword. The compiler distinguishes between constants identifiers and other identifiers (like polynomial identifiers) via the percent `%` symbol. Henceforth, constant identifiers should be preceded by the percent symbol.

```
// Filename: config.pil

constant %N = 2**10;
```

In order to relate both files, the `include` reserved keyword must be invoked.

```
// Filename: multiplier.pil

include "config.pil";

namespace Multiplier(%N);

// Constant Polynomials
pol constant RESET;

// Committed Polynomials
pol commit freeIn;
pol commit out;

// Intermediate Polynomials
pol carry = out*freeIn;

// Constraints
out' = RESET*freeIn + (1-RESET)*carry;
```

1.4 Cyclic Constraints

Since execution traces have finite length N, there is one implicit complexity in the design of programs with PIL: constraints should be satisfied over every element of a subgroup $G = \langle g \rangle$ of $\mathbb{F}^*$ of size N. This means that the description (in terms of constraints) of a program is not correct if the appropriate constraints are not satisfied in every row transition. This is because the polynomials defined from the columns of the execution trace are constructed by interpolation at G. In particular, constraints containing polynomials using the prime notation should remain true in the transition from the last row to the first row because $g \cdot g^N = g \cdot 1 = g$ (by Lagrange's theorem) and therefore:

$$p'(g^N) = p(g \cdot g^N) = p(g),$$

which is the first value of the column defined by the polynomial p.

This is an important aspect that has to be taken care of when designing the set of constraints of a program with PIL. If there is some constraint that is not satisfied in the last transition, one normally overcomes this problem by the inclusion of additional polynomials that solve this issue. For example, consider the following PIL code:

```
namespace CyclicExample(4);
pol commit a, b;

(a+1)*a*(a-1) = 0;
b' = b+a;
```

and the following valid execution trace:

row	a	b
1	1	1
2	0	2
3	-1	2
4	1	1

Observe that the polynomial $\texttt{a}$ only takes the values $0, 1$ or -1 (equivalently, $p-1$) and this is precisely captured by the constraint:

$$(\texttt{a}+1) \cdot \texttt{a} \cdot (\texttt{a}-1) = 0,$$

which is satisfied for each row $i \in [4]$.

On the other side, the second constraint $\texttt{b}' = \texttt{b} + \texttt{a}$ is satisfied for every row except for the last, because:

$$\texttt{b}'(g^4) = \texttt{b}(1) = 1 \neq 2 = 1 + 1 = \texttt{b}(g^4) + \texttt{a}(g^4), \tag{1}$$

where g is the generator of the group G of size 4.

The idea to overcome this problem is to add a constant column $\texttt{SEL}$ that is 1 in every row except for the last one and introduce it to the previous constraint as follows:

$$\texttt{b}' = \texttt{SEL} \cdot (\texttt{b} + \texttt{a}) + (1 - \texttt{SEL}).$$

Notice that now the right-hand side of Eq. (1) is equal to $\texttt{SEL}(g^4) \cdot (\texttt{b}(g^4) + \texttt{a}(g^4)) + (1 - \texttt{SEL}(g^4)) = 1$.

The valid PIL is the following:

```
namespace CyclicExample(4);

pol commit a, b;
pol constant SEL;
pol carry = (a+1)*a;

carry*(a-1) = 0;
b' = SEL*(b+a) + (1-SEL);
```

1.5 Inclusion Arguments

There is a subtlety in the Multiplier example of Section 1.2 that is worth commenting on at this point. It is assumed that columns can contain field elements but, it may be the case where one needs to restrict the size of the columns' values to a certain number of bits. Henceforth, it is important to develop a strategy to control both underflows and overflows. In this section, we will design a program (and its corresponding PIL) to verify the addition of two integers of a size fitting in exactly 2 bytes. That is, any input to the addition will be considered invalid if it is not an integer in the range $[0, 65535]$.

Of course, there is plenty of ways to arithmetize this program. However, in this section, we are going to use a method based on inclusion arguments. The overall idea is to reduce 2-byte additions to 1-byte additions. Specifically, the program will consist of two input polynomials `a` and `b` where each one will, at each row, introduce a single byte (equivalently, an integer in the range $[0, 255]$) for each operand of the sum, starting with the less significant byte. Hence, every two rows, a new addition will be checked.

row	a	b	operation
1	0x11	0x22	*undefined*
2	0x30	0x40	0x3011 + 0x4022
3	0xff	0xee	*undefined*
4	0x00	0xff	0x00ff + 0xffee

The output of the addition between words of 1-byte can not be stored in a single column that only accepts words of 1-byte, since an overflow may appear. Hence, the result of the addition will be split into two columns `carry` and `add`, accepting words of exactly 1-byte, to completely store the result so that a correct addition between bytes can be defined as:

$$\mathtt{a} + \mathtt{b} = \mathtt{carry} \cdot 2^8 + \mathtt{add}. \tag{2}$$

Table 1 shows an example of a valid execution trace for this program. Note that the result of operating each of the 2-bytes words can be obtained by simply grabbing the value in the last `carry` value and each of the last two `add`'s values in decreasing order. For example, 0x3011 + 0x4022 = 0x007033, which can be recovered from the blue cells below. The same applies for 0x00ff + 0xffee = 0x0100ed, which can be recovered similarly from the pink cells.

Table 1: TBD

row	a	b	carry	add
1	0x11	0x22	0x00	0x33
2	0x30	0x40	0x00	0x70
3	0xff	0xee	0x01	0xed
4	0x00	0xff	0x01	0x00

One can notice that constraint (2) is not satisfied at row 4, because the `carry`'s value that is raised in the third row must be introduced into the 1-byte addition at row 4. In this way, we can link the two-byte sums that compose 2-byte additions. There is no direct way to introduce the previous row of a column in PIL (in contrast with the next row, which we can invoke with the single quote operator '). The idea then is to introduce another committed polynomial called `prevCarry` which will contain a shifted version of the value in the `carry` polynomial. More specifically, we add the following constraint to ensure that `prevCarry` is correctly defined:

$$\texttt{prevCarry}' = \texttt{carry} \tag{3}$$

row	a	b	prevCarry	carry	add
1	0x11	0x22	0x01	0x00	0x33
2	0x30	0x40	0x00	0x00	0x70
3	0xff	0xee	0x00	0x01	0xed
4	0x00	0xff	0x01	0x01	0x00

Now we face another problem: the previous carry must not affect two different 2-byte additions as per constraint (3). That is, non-related operations should not be linked via carries. Henceforth, a constant polynomial called `RESET` will be introduced to break the present connection between different additions. The `RESET` polynomial will be a binary-ranged polynomial which will be 1 at the beginning of every 2 byte addition. Elsewhere, `RESET` will evaluate to 0.

row	a	b	prevCarry	carry	add	RESET
1	0x11	0x22	0x01	0x00	0x33	1
2	0x30	0x40	0x00	0x00	0x70	0
3	0xff	0xee	0x00	0x01	0xed	1
4	0x00	0xff	0x01	0x01	0x00	0

Following this logic, we can now derive the final (and accurate) constraint:

$$\mathsf{a} + \mathsf{b} + (1 - \mathsf{RESET}) \cdot \mathsf{prevCarry} = \mathsf{carry} \cdot 2^8 + \mathsf{add}. \tag{4}$$

The PIL program of the previous example can be written as follows:

```
include "config.pil";

namespace TwoByteAdd(%N);

pol constant RESET;
pol commit a, b;
pol commit carry, prevCarry, add;

prevCarry' = carry;
a + b + (1-RESET)*prevCarry = carry*2**8 + add;
```

Similarly to the Multiplier example of Section 1.2, it is worth mentioning that only by changing the `RESET` polynomial (but not the PIL itself), it is possible to arithmetize a program being able to verify generic n-bytes additions. In such case, `RESET` will be 1 for every n row and 0 otherwise.

Up to this point, one can think that Constraint (4) restricts a sound representation of the program. However, since we are working over a finite field, it is not. For example, the following execution trace is valid, as it satisfies all the constraints, but does not correspond to a valid computation:

row	a	b	prevCarry	carry	add	RESET
1	0x11	0x22	0x01	$p \cdot 2^{-8}$	0x33	1
2	0x30	0x40	$p \cdot 2^{-8}$	0x00	$\texttt{0x70} + p \cdot 2^{-8}$	0
3	0xff	0xee	0x00	$\texttt{0x01} + p \cdot 2^{-8}$	0xed	1
4	0x00	0xff	$\texttt{0x01} + p \cdot 2^{-8}$	0x01	$\texttt{0x00} + p \cdot 2^{-8}$	0

Figure 3: Concrete example showing how to cheat the current program restrictions.

Moreover, nothing is preventing the introduction of more bytes in either column ør , which breaks the intention of this program, specially designed to deal only with byte-sized columns. Enforcing that all the evaluations of committed polynomials are correct is strictly necessary. As explained at the beginning of this section, an inclusion argument will be used.

Definition 1 (Inclusion Argument)**.** Given two vectors $a = (a_1, \dots, a_n) \in \mathbb{F}_p^n$ and $b = (b_1, \dots, b_m) \in \mathbb{F}_p^m$, we say that a *is contained in* b if for all $i \in [n]$, there exists a $j \in [m]$ such that $a_i = b_j$. In words, if we see a and b as multisets and then we reduce them to sets (by removing the multiplicity), then a is contained in b if a is a subset of b.

Moreover, we say that a protocol $(\mathcal{P}, \mathcal{V})$ is an *inclusion argument* if the protocol can be used by $\mathcal{P}$ to prove to $\mathcal{V}$ that one vector is contained in another vector.

In the PIL context, the implemented inclusion argument is the inclusion argument provided here [GW20], with the "alternating method" provided in [PFM+22] and can be invoked via the in keyword. Specifically, given two columns a, b, we can declare an inclusion argument between them using the syntax {a} in {b}, where a, b does not necessarily need to be defined in different programs. See for example:

```
include "config.pil";

namespace A(%N);
pol commit a, b;

{a} in {b};

namespace B(%N);
pol commit a, b;

{a} in {Example1.b};
```

A valid execution trace (with $N = 4$) for the previous example is shown in Table 4.

Table 2: Execution traces for programs A and B.

Program A		**Program B**	
a	b	a	b
3	1	1	6
4	2	1	7
2	3	1	8
3	4	4	9

In PIL we can also write inclusion arguments not only over single columns but over multiple columns. That is, given two subsets of committed columns $\texttt{a}_1, \dots, \texttt{a}_m$ and $\texttt{b}_1, \dots, \texttt{b}_m$ of some program(s) we can write $\{\texttt{a}_1, \dots, \texttt{a}_m\}$ in $\{\texttt{b}_m, \dots, \texttt{b}_m\}$ to denote that the rows generated by columns $\mathtt{b}_m, \dots, \mathtt{b}_m$ are included (as sets) into the rows generated by columns $\{\mathtt{a}_1, \dots, \mathtt{a}_m\}$. A natural application for this generalization is showing that a set of columns in a program repeatedly computes, probably with the same pair of inputs/outputs, an operation such as AND, where the correct AND operation is carried on a distinct program (see the following example).

```
include "config.pil";

namespace Main(%N);
pol commit a, b, c;

{a,b,c} in {in1,in2,xor};

namespace AND(%N);
pol constant in1, in2, and;
```

Following with the previous example, the idea will be to construct 4 new constant polynomials BYTE_A, BYTE_B, BYTE_CARRY and BYTE_ADD which contain all possible byte additions. The execution trace of these polynomials can be constructed as follows:

row	BYTE_A	BYTE_B	BYTE_CARRY	BYTE_ADD
1	0x00	0x00	0x00	0x00
2	0x00	0x01	0x00	0x01
3	0x00	0x02	0x00	0x02
⋮	⋮	⋮	⋮	⋮
256	0x00	0xff	0x00	0xff
257	0x01	0x00	0x00	0x01
258	0x01	0x01	0x00	0x02
⋮	⋮	⋮	⋮	⋮
65535	0xff	0xfe	0x01	0xfd
65536	0xff	0xff	0x01	0xfe

Figure 4: Summary of the values for the constant polynomials in the TwoByteAdd program.

Recall that enforcing constraints between these polynomials will not be needed since they are constant and therefore, publicly known.

Ensuring that the tuple $(\mathtt{a}, \mathtt{b}, \mathtt{carry}, \mathtt{add})$ is contained in the previous table via an inclusion argument will ensure a sound description of the program. The inclusion constraint is not only ensuring that all the values are single bytes, but also checking that the addition is correctly computed. Consequently, none of the rows of Table 3 is contained in the previous table, marking them as non-valid rows. Of course, this will introduce some redundancy into the PIL, because the byte operation is being checked twice (one with the polynomial constraint and the other with the inclusion argument). However, we can not simply drop the polynomial constraint because it is necessary for linking rows belonging to the same addition.

The following line of code completes the PIL for the TwoByteAdd program:

```
{a, b, carry, add} in {BYTE_A, BYTE_B, BYTE_CARRY, BYTE_ADD};
```

To sum up, the following PIL program correctly describes the TwoByteAdd program:

```
include "config.pil";

namespace TwoByteAdd(%N);

pol constant BYTE_A, BYTE_B, BYTE_CARRY, BYTE_ADD;
pol constant RESET;
pol commit a, b;
pol commit carry, prevCarry, add;

prevCarry' = carry;
a + b + (1 - RESET)*prevCarry = carry*2**8 + add;

{a, b, carry, add} in {BYTE_A, BYTE_B, BYTE_CARRY, BYTE_ADD};
```

Compiling this .pil file, we get the following debugging message:

```
Input Pol Commitmets: 5
Q Pol Commitmets: 0
Constant Pols: 5
Im Pols: 0
plookupIdentities: 1
permutationIdentities: 0
connectionIdentities: 0
polIdentities: 3
```

Observe that plookupIdentities counts the number of inclusion arguments used in the PIL program (one in our example).

1.5.1 Avoiding Redundancy

Further modifications can be added to avoid redundancy in the PIL. This can be achieved by introducing another constant polynomial BYTE_PREVCARRY. In this case, the constant polynomials table formed by the polynomials BYTE_A, BYTE_B, BYTE_PREVCARRY, BYTE_CARRY and BYTE_ADD should be generated by iterating among all the possible combinations of the tuple (BYTE_A, BYTE_B, BYTE_PREVCARRY) and computing BYTE_CARRY and BYTE_ADD accordingly in each of the combinations. The table only becomes twice bigger because BYTE_PREVCARRY is binary. A summary of how the table looks like with the new changes can be found in Figure 5.

In addition, recall that we only have to take into account prevCarry whenever RESET is 0. PIL is flexible enough to consider this kind of situation involving Plookups. To introduce this requirement, the inclusion check can be modified as follows:

```
{a, b, (1 - RESET)*prevCarry, carry, add} in {BYTE_A, BYTE_B,
    BYTE_PREVCARRY, BYTE_CARRY, BYTE_ADD};
```

With this modification, the PIL program becomes:

row	BYTE_A	BYTE_B	BYTE_PREVCARRY	BYTE_CARRY	BYTE_ADD
1	0x00	0x00	0x00	0x00	0x00
2	0x00	0x01	0x00	0x00	0x01
3	0x00	0x02	0x00	0x00	0x02
⋮	⋮	⋮	⋮	⋮	⋮
256	0x00	0xff	0x00	0x00	0xff
257	0x01	0x00	0x00	0x00	0x01
258	0x01	0x01	0x00	0x00	0x02
⋮	⋮	⋮	⋮	⋮	⋮
65535	0xff	0xfe	0x00	0x01	0xfd
65536	0xff	0xff	0x00	0x01	0xfe
65537	0x00	0x00	0x01	0x00	~~0x00~~
65538	0x00	0x01	0x01	0x00	0x02
65539	0x00	0x02	0x01	0x00	0x03
⋮	⋮	⋮	⋮	⋮	⋮
65792	0x00	0xff	0x01	0x01	0x00
65793	0x01	0x00	0x01	0x00	0x02
65794	0x01	0x01	0x01	0x00	0x03
⋮	⋮	⋮	⋮	⋮	⋮
131071	0xff	0xfe	0x01	0x01	0xfe
131072	0xff	0xff	0x01	0x01	0xff

Figure 5: Summary of the values for the constant polynomials adding BYTE_PREVCARRY.

```
include "config.pil";

namespace TwoByteAdd(%N);

pol constant BYTE_A, BYTE_B, BYTE_PREVCARRY, BYTE_CARRY, BYTE_ADD;
pol constant RESET;
pol commit a, b;
pol commit carry, prevCarry, add;

prevCarry' = carry;

{a, b, (1 - RESET)*prevCarry, carry, add} in {BYTE_A, BYTE_B,
    BYTE_PREVCARRY, BYTE_CARRY, BYTE_ADD};
```

1.6 Connecting Programs

One of the core features of PIL is that it allows a *modular design* of its programs. By modular we mean the ability to split the design of a program M in multiple small programs such that the (proper) combination of the small programs lead to M. Without this feature, one would need to combine the logic of multiple pieces in a single design and, as a consequence, a lot of redundant polynomials would be included in the design. Moreover, the resulting big design would be difficult to validate and test.

Through this section, the modularity feature will be shown using the following example. Suppose that we want to design a program to verify the operation $a \cdot \overline{a}$, where a is a 4-bits integer and $\overline{a}$ is defined as the integer obtained by negating each of the bits of a (i.e., $\overline{a}$ is the bitwise negation of a). The difficulty here is that bitwise operations are difficult to describe when working directly with chunks of 4-bits. Then, the idea will be to split them into bits in another program, allowing us to check the operations in a trivial way.

The main program will consist in 3 columns: `a`, `neg_a` and `op`. The column `a` will contain at each row the integer a of the computation $a \cdot \overline{a}$. The columns `neg_a` and `op` will contain $\overline{a}$ and $a \cdot \overline{a}$, respectively. Table 3 represents a valid execution trace of the program.

Table 3: Concrete example of the program that validates negated strings of bits.

`row`	`a`	`neg_a`	`op`
1	1101	0010	00011010
2	0100	1011	00101100
3	1111	0000	00000000
4	1000	0111	00111000

First, there is the need to enforce that each of the inputs is effectively a 4-bits integer (that is, an integer in the range $[0, 15]$). This can be enforced via an inclusion argument. Specifically, this argument enforces that all the values of a vector belong to a certain (publicly known) range. For this reason, this family of inclusion arguments is often referred to as *range checks*. The PIL code for the range check for column `a` is as follows:

```
include "config.pil";

namespace Global(%N);
pol constant BITS4;

namespace Main(%N);
pol commit a, neg_a, op;

a in Global.BITS4;
```

There are two important remarks about the former PIL code. First, `BITS4` is a polynomial containing each one of the possible 4-bits integers. Any order can be chosen to construct it because inclusion checks do not care about the ordering. Second, observe that `BITS4` is called from a different namespace where it is defined. The syntax `Namespace.polynomial` can be used to access other's namespaces polynomials. The idiomatic way of proceeding is to put different namespaces in separated files and then use the `include` keyword to "connect" them:

```
// Filename: global.pil

include "config.pil";

namespace Global(%N);
pol constant L1;
pol constant BITS4;
```

```
// Filename: main.pil

include "config.pil";
include "global.pil"

namespace Main(%N);
pol commit a, neg_a, op;

a in Global.BITS4;
```

As mentioned before, working with 4 bits directly would be difficult. Hence, another program that works bitwise will be designed. This program will contain a column `bits` that will store the single bits that shape each of the integers present in a in expressed as little-endian. Similarly, a column `nbits` will contain the negation of each one of the previous bits. For a concrete example of the execution trace, see Table 6.

row	bits	nbits	FACTOR
1	1	0	1
2	0	1	2
3	1	0	2^2
4	1	0	2^3
5	0	1	1
6	0	1	2
7	1	0	2^2
8	0	1	2^3
$\vdots$	$\vdots$	$\vdots$	$\vdots$

Figure 6: Relationship between `bits` and `nbits` columns.

Therefore, since $\overline{\texttt{bits}} = \texttt{nbits}$ and $\texttt{bits}, \texttt{nbits} \in \{0, 1\}$, we can express the relation between the columns `bits` and `nbits` as:

$$\texttt{bits} + \texttt{nbits} = 1.$$

The idea to connect the Main program with the Negation program will be to construct a and $\overline{a}$ from the given bits for each freely input integer a. Therefore, the inclusion argument that verifies that the tuple $(a, \overline{a})$ from the Main program is included in the Negation program will prove the fact that $\overline{a}$ is correctly constructed. To do that we will need a constant polynomial called `FACTOR` (see Table 6) that will allocate the bits in their correct position.

At the same time, a constant polynomial `RESET` is necessary to allow us to reset the generation of the columns `a` and `neg_a` from the bits of `bits` and `nbits` respectively every 4 row. Observe that cyclic behavior is ensured in this situation since 4 divide $N = 2^{10}$ (more generally, since N should be a power of 2, this requirement will also be satisfied). The constraints that should be added to PIL to describe this generation are the following ones:

$$\begin{aligned}
&\texttt{a'} = \texttt{FACTOR'} \cdot \texttt{bits'} + (1 - \texttt{RESET}) \cdot a \\
&\texttt{neg\_a'} = \texttt{FACTOR'} \cdot \texttt{nbits'} + (1 - \texttt{RESET}) \cdot \texttt{neg\_a};
\end{aligned}$$

Figure 7 shows a complete example of what the execution trace of the Negation program looks like. Observe that at each row the previous constraints are being satisfied thanks to the introduction of the `RESET` polynomial.

Now we can write the PIL file that describes a correct execution of the Negation program as follows:

row	bits	nbits	FACTOR	a	neg_a	RESET
1	1	0	1	1	0	0
2	0	1	2	01	10	0
3	1	0	2^2	101	010	0
4	1	0	2^3	1101	0010	1
5	0	1	1	0	1	0
6	0	1	2	00	11	0
7	1	0	2^2	100	011	0
8	0	1	2^3	0100	1011	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Figure 7: Complete example of the Negation program.

```
// Filename: negation.pil

namespace Negation(%N);
pol commit bits, nbits;
pol commit a, neg_a;
pol constant FACTOR, RESET;

bits*(1-bits) = 0;
nbits*(1-nbits) = 0;

bits + nbits - 2*bits*nbits = 1;

a' = FACTOR'*bits' + (1 - RESET)*a;
neg_a' = FACTOR'*nbits' + (1 - RESET)*neg_a;
```

Observe that binary checks for the columns `bits` and `nbits` have been added because we need to ensure that both of them are, in fact, bits. Now, we will connect the Main and the Negation program via an inclusion check, adding the following line into the `Main` PIL's namespace:

```
{a, neg_a} in {Negation.a, Negation.neg_a};
```

Although this is not highly important in this example, PIL also allows the user to add selectors in the inclusion checks. This feature gives huge malleability as it allows the user to check inclusions between smaller subsets of rows of the execution trace. For example

```
{a, neg_a} in Negation.RESET {Negation.a, Negation.neg_a}
```

will enforce checking that a tuple $(a, \overline{a})$ is contained in the columns `a`, `neg_a` in a row having specifically the `RESET` value equal to 1. However, as being said, this introduces redundancy in the PIL because of the design of the program itself. The same is possible from the other side, adding a selector `SEL` present in the Main program:

```
sel {a, neg_a} in {Negation.a, Negation.neg_a}
```

The former is crucial in PIL because it will be mainly used in a situation where inclusions must only satisfied in a subset of rows and therefore, including the inclusion check in PIL without any selector will cause the it not to be satisfied in the complementary rows subset. Moreover, we will see later on that this is of huge importance to improve proving performance. This is possible because adding correct selectors will make the correspondence between rows bijective, allowing us to substitute the inclusion argument with a permutation argument (see Section 1.8), which is far more efficient. We can also use a combination of selectors, one for each side:

```
sel {a, neg_a} in Negation.RESET {Negation.a, Negation.neg_a}
```

Up to this point we have created program (called Main) that uses another program (called Negation) to validate that the negation of a specific column `a` is well constructed. However, we still need to validate the product of the columns `a` and `neg_a`. We can simply introduce the constraint

$$\mathtt{a} \cdot \mathtt{neg\_a} = \mathtt{op}$$

into the PIL of the Main program. Nonetheless, since we are exemplifying how connections among several programs work, we can also use the previously constructed first version of the Multiplier program to do that. Adding the following line of code does the job.

```
{a, neg_a, op} in {Multiplier.freeIn1, Multiplier.freeIn2,
    Multiplier.out};
```

To sum up, the PIL code of all the newly developed programs can be found below:

```
// Filename: "global.pil"

include "config.pil";

namespace Global(%N);
pol constant BITS4;
```

```
// Filename: "main.pil"

include "global.pil";
include "multiplier.pil";
include "negation.pil";
include "config.pil";

namespace Main(%N);
pol commit a, neg_a, op;

a in Global.BITS4;

{a, neg_a} in {Negation.a, Negation.neg_a};
{a, neg_a, op} in {Multiplier.freeIn1, Multiplier.freeIn2,
    Multiplier.out};
```

```
// Filename: "negation.pil"

include "config.pil";

namespace Negation(%N);
pol commit bits, nbits;
pol commit a, neg_a;
pol constant FACTOR, RESET;

bits*(1-bits) = 0;
nbits*(1-nbits) = 0;

bits + nbits - 2*bits*nbits = 1;

a' = FACTOR'*bits' + (1 - RESET)*a;
neg_a' = FACTOR'*nbits' + (1 - RESET)*neg_a;
```

Having all the `.pil` files in the same directory we can compile `main.pil` in order to obtain the following message

```
Input Pol Commitments: 10
Q Pol Commitments: 0
Constant Pols: 3
Im Pols: 0
plookupIdentities: 3
permutationIdentities: 0
connectionIdentities: 0
polIdentities: 6
```

1.7 Public Values

Public values are defined to be values of committed polynomials that are known to both the prover and the verifier as part of the arithmetization process. For example, if the prover is claiming to know the output of a certain computation, then its arithmetization would lead to the inclusion of a public value to some of the polynomials representing such computation. In this section, we will build a PIL program that arithmetizates a Fibonacci sequence making use of public values.

Modular Fibonacci Sequence Imagine one wants to prove knowledge of the first two terms F_1 and F_2 of a Fibonacci sequence $(F_n)_{n \in \mathbb{N}}$ whose 1024-th therm F_{1024} is:

$$F_{1024} = 180312667050811804,$$

modulo the prime $p = 2^{64} - 2^{32} + 1$. The witness (that is, the input kept private by the prover) is $F_1 = 2$ and $F_2 = 1$.

We can arithmetizate the modular Fibonacci sequence with 3 columns:

- Two committed polynomials `a` and `b` will keep track of the sequence elements. We naturally obtain the following constraints between `a` and `b`:

$$\begin{aligned} \mathtt{a}' &= \mathtt{b} \\ \mathtt{b}' &= \mathtt{a} + \mathtt{b} \end{aligned}$$

- The constant polynomial `ISLAST` will be used to ensure the previous constraints are valid even in the last row, i.e., $\mathtt{ISLAST}(g^i) = 0$ if $i \in [N-1]$ and 1 at g^N. Note that this last row is precisely the point at which it is ensured whether the claimed 1024-th term is correct or not. With the introduction of ISLAST the previous constraints become:

$$(1 - \texttt{ISLAST}) \cdot (\texttt{a'} - \texttt{b}) = 0$$
$$(1 - \texttt{ISLAST}) \cdot (\texttt{b'} - \texttt{a} - \texttt{b}) = 0$$

Based on the previous description, the PIL for the Fibonacci sequence is as follows:

```
// Filename: "fib.pil"

namespace Fibonacci(2**10);
pol constant ISLAST;
pol commit a, b;

(1-ISLAST) * (a' - b) = 0;
(1-ISLAST) * (b' - a - b) = 0;
ISLAST * (a - 180312667050811804) = 0;
```

Notice that we have hardcoded the 1024-th Fibonacci term 180312667050811804 in the PIL description so that if we would like to use the same PIL for any other witness F_1, F_2 or any other Fibonacci term as the output of the original claim, then we would need to modify the PIL to cover the new parameters. To avoid this issue, we will introduce *public values*. With public values, the prover will be able to change the witness F_1, F_2 or the Fibonacci output without making any changes in the PIL program.

We introduce the last term of a as a public value in PIL using the following syntax, where the integer inside the parenthesis corresponds to the row's number where the term is located:

```
public result = a(%N-1);
```

Then, the compiler distinguishes between public values identifier and other identifiers via the colon :. So that to refer to result we will write :result. The updated PIL file is as follows:

```
// Filename: "fib.pil"

include "config.pil";

namespace Fibonacci(%N);
pol constant ISLAST;
pol commit a, b;

public result = a(%N-1);

(1-ISLAST) * (a' - b) = 0;
(1-ISLAST) * (b' - a - b) = 0;
ISLAST * (a - :result) = 0;
```

1.8 Permutation Arguments

In this section, a new kind of constraint that can be used in PIL is introduced: *permutation arguments*.

Definition 2 (Permutation Argument)**.** Given two vectors $a = (a_1, \dots, a_n)$ and $b = (b_1, \dots, b_n)$ in $\mathbb{F}^n$, we say that *a and b are a permutation of each other* if there exists a permutation $\sigma\colon [n] \to [n]$ such that $a = \sigma(b)$, where $\sigma(b)$ is defined by:

$$\sigma(b) := (b_{\sigma(1)}, \dots, b_{\sigma(n)}).$$

Moreover, we say that a protocol $(\mathcal{P}, \mathcal{V})$ is a *permutation argument* if the protocol can be used by $\mathcal{P}$ to prove to $\mathcal{V}$ that two vectors in $\mathbb{F}^n$ are a permutation of each other.

Observe that, unlike inclusion arguments, the two vectors subject to a permutation argument must have the same length. In the PIL context, the Plookup permutation argument between two columns $\mathtt{a}, \mathtt{b}$ can be declared using the keyword `is` using the syntax `{a} is {b}`, where $\mathtt{a}, \mathtt{b}$ does not necessarily need to be defined in different programs.

```
include "config.pil";

namespace A(%N);
pol commit a, b;

{a} is {b};

namespace B(%N);
pol commit a, b;

{a} is {Example1.b};
```

A valid execution trace (with a number of rows $N = 4$) for the previous example is shown in Table 4.

Table 4: Execution traces for programs A and B.

Program A		**Program B**	
a	b	a	b
8	2	3	6
1	3	8	7
2	8	1	8
3	1	2	9

In PIL we can also write permutation arguments not only over single columns but over multiple columns. That is, given two subsets of committed columns $\mathtt{a}_1, \dots, \mathtt{a}_m$ and $\mathtt{b}_1, \dots, \mathtt{b}_m$ of some program(s) we can write $\{\mathtt{a}_1, \dots, \mathtt{a}_m\}$ `is` $\{\mathtt{b}_m, \dots, \mathtt{b}_m\}$ to denote that the rows generated by columns $\mathtt{b}_m, \dots, \mathtt{b}_m$ are a permutation of the rows generated by columns $\{\mathtt{a}_1, \dots, \mathtt{a}_m\}$. A natural application for this generalization is showing that a set of columns in a program is precisely a commonly known operation such as XOR, where the correct XOR operation is carried on a distinct program (see the following example).

```
include "config.pil";

namespace Main(%N);
pol commit a, b, c;

{a,b,c} is {in1,in2,xor};

namespace XOR(%N);
pol constant in1, in2, xor;
```

However, the functionality is further extended by the introduction of selectors in the sense that one can still write a permutation argument even tho it is not satisfied over the entire trace of a set of columns but over a subset of it. Suppose that we are given the following execution traces:

Table 5: Execution traces for programs A and B.

Program A						
...	SEL	a	b	...	c	...
...	0	8	5	...	1	...
...	1	1	2	...	7	...
...	0	2	9	...	6	...
...	1	3	5	...	2	...

Program B						
...	sel	d	e	...	f	...
...	0	3	3	...	4	...
...	0	7	9	...	3	...
...	1	5	3	...	2	...
...	0	2	9	...	6	...
...	1	2	1	...	7	...
...	0	9	6	...	6	...

Notice that columns $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ of the program A and columns $\{\mathtt{e}, \mathtt{d}, \mathtt{f}\}$ of the program B are a permutation of each other only over a subset of the trace. To still achieve a valid permutation argument over such columns, we have introduced a committed column `sel` set to 1 in rows where we want to enforce the permutation argument. Therefore, the permutation argument will only be valid if (a) `sel` is correctly computed and (b) the subset of rows chosen by `sel` in both programs shows a permutation. The corresponding PIL of the previous programs can be written as follows:

```
namespace A(4);
pol commit a, b, c;
pol commit sel;

sel {a, b, c} is B.sel {B.e, B.d, B.f};

namespace B(6);
pol commit d, e, f;
pol commit sel;
```

As a final and important remark, the `sel` column should be turned on the same amount of times in both programs. Otherwise, a permutation can not exist between any of the columns since the resulting vectors would be of different lengths. This allows us to use this kind of argument even if both execution traces do not contain the same amount of rows.

1.9 Connection Arguments

In this section, a new kind of constraint that can be used in PIL is introduced: *connection arguments*. Suppose that we are given a vector $a = (a_1, \dots, a_n)$ and a partition S of $[n]$.

Definition 3 (Connection Argument)**.** Given a vector $a = (a_1, \dots, a_n) \in \mathbb{F}^n$ and a partition $\S = \{S_1, \dots, S_t\}$ of $[n]$, we say that *a copy-satisfy* $\S$ if for each $S_k \in \S$ we have that $a_i = a_j$ whenever $i, j \in S_k$, with $i, j \in [n]$ and $k \in [t]$.

Moreover, we say that a protocol $(\mathcal{P}, \mathcal{V})$ is a *connection argument*[1] if the protocol can be used by $\mathcal{P}$ to prove to $\mathcal{V}$ that a vector copy-satisfies a partition of $[n]$.

Let us show a specific example. Let $\S = \{\{2\}, \{1, 3, 5\}, \{4, 6\}\}$ be the specified partition of $[6]$. Observe the two columns depicted below:

[1]We refer to it as "connection" instead of "copy-satisfaction" since we use this argument in PIL in a more general way that the way it is used in the paper where it was originally defined [GWC19].

a	b
🟦 3	🟦 3
9	9
🟦 3	🟥 7
1	1
🟦 3	🟦 3
1	1

Figure 8: Two columns: one that copy-satisfies §and other that does not.

The vector a copy-satisfies §because $a_1 = a_3 = a_5 = 3$ and $a_4 = a_6 = 1$. Observe that, since $\{2\}$ is a singleton in §, a_2 is not related to any other element in a. However, the vector b does not copy-satisfies §because $b_1 = b_5 = 3 \neq 7 = b_3$.

In the programs' context, connection arguments can be easily written in PIL by introducing a column associated with the chosen partition as in [GWC19]. Recall that column values are evaluations of a polynomial at $G = \langle g \rangle$, where N is the length of the execution trace. Suppose we are given a polynomial a and partition § and we want to write in PIL a constraint attesting to the copy-satisfiability of a concerning §. We first construct a permutation $\sigma\colon [n] \to [n]$ such that for each set $S_i \in §$, $\sigma(§)$ contains a cycle going over all elements of S_i. In the previous example, we would have $\sigma = (5, 2, 1, 6, 3, 4)$. Then, we construct a polynomial S_a that encodes σ in the exponent of g, that is:

$$S_a(g^i) = g^{\sigma(i)},$$

for $i \in [n]$.

In the PIL context, the previous connection argument between a column a and a column SA encoding the values of S_a can be declared using the keyword connect using the syntax {a} connect {SA}.

```
include "config.pil";

namespace Connection(%N);
pol commit a;
pol constant SA;

{a} connect {SA};
```

A valid execution trace for this example was shown in Figure 8.

Remark 1. The column SA does not necessarily need to be declared as a constant polynomial. The connection argument will still hold if it is declared as committed.

Connection arguments can be extended to several columns by encoding each column with a "part" of the permutation. Informally, the permutation is now able to span across the values of each of the involved polynomials in a way that the cycles formed in the permutation must contain the same value.

Definition 4 (Multi-Column Copy-Satisfiability)**.** Given vectors $a_1, \dots, a_k$ in $\mathbb{F}^n$ and a partition $§ = \{S_1, \dots, S_t\}$ of $[kn]$, we say that $a_1, \dots, a_k$ *copy-satisfy* § if for each $S_m \in §$ we have that $a_{\ell_1, i} = a_{\ell_2, j}$ whenever $i, j \in S_m$, with $i, j \in [n]$, $\ell_1, \ell_2 \in [k]$ and $m \in [t]$.

For example, say that we have $§ = \{\{1\}, \{2, 3, 4, 9\}, \{5\}, \{6\}, \{7, 10\}, \{8, 11\}, \{12\}\}$. Then, Table 6 depicts the execution trace of three columns a, b, c that copy-satisfies §.

Table 6: An execution trace subject to a connection argument.

a	b	c
1	2	3
3	4	5
3	5	6
3	6	7

We reduce this problem to the problem with one column in the sense that we think of the permutation σ as applied to the concatenation of column `a`, then `b` and finally `c`. So, the permutation σ that makes `a`, `b`, `c` copy-satisfy § is $(1, 9, 2, 3, 5, 6, 10, 11, 4, 7, 8, 12)$. In this case we construct polynomials S_a, S_b, S_c such that:

$$S_a(g^i) = g^{\sigma(i)}, \quad S_b(g^i) = k_1 \cdot g^{\sigma(n+i)}, \quad S_c(g^i) = k_2 \cdot g^{\sigma(2n+i)}$$

where we have introduced $k_1, k_2 \in \mathbb{F}$ as a way of obtaining more elements (in a group G of size n) to be able to correctly encode the $[3n] \to [3n]$ permutation σ. For more details on this encoding see [GWC19].

The table below shows how to compute the polynomials SA, SB and SC encoding the permutation of the previous example:

a	b	c	SA	SB	SC
1	2	3	1	k_1	g^3
3	4	5	k_2	$k_1 \cdot g$	$k_1 \cdot g^2$
3	5	6	g	$k_2 \cdot g$	$k_1 \cdot g^3$
3	6	7	g^2	$k_2 \cdot g^2$	$k_2 \cdot g^3$

Figure 9: Multi-column connection argument's valid execution trace.

The PIL of this example can be easily written as follows:

```
include "config.pil";

namespace Connection(%N);
pol commit a, b, c;
pol constant SA, SB, SC;

{ a, b, c } connect { SA, SB, SC };
```

1.9.1 PlonK in PIL

As a use case example, we can implement PlonK verification using connection arguments. Suppose that we are given a PlonK-like circuit C. The definition of the circuit defines a set of preprocessed polynomials `QL`, `QR`, `QM`, `QO` and `QC` describing all the PlonK gates (that is, interpolating the values of the PlonK selectors at each of the gates in a selected order), as well as a triple of connection polynomials `SA`, `SB` and `SC` specifying the copy-constraints that are needed to be satisfied. The polynomials `SA`, `SB` and `SC` are constructed as before. We provide a concrete but small example of what this circuit-to-trace translation looks like. The following PlonK circuit shown in Figure 10 raises the execution trace depicted blow.

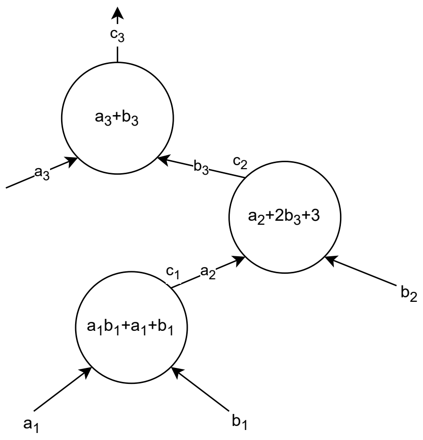

Figure 10: Example of a $\mathcal{P}$lon$\mathcal{K}$-like circuit.

Observe all the values below depend only on the shape of the circuit itself, so if the circuit does not change, this polynomial should not be recomputed. Check that the SA, SB and SC polynomials are constructed so that we can verify the copy-constraints $c_2 = b_3$ and $c_1 = a_2$ present in the circuit. Observe that equally painted cells have their values swapped, as explained before in the construction of the connection S polynomials.

QL	QR	QM	QO	QC	SA	SB	SC
1	1	1	-1	0	1	k_1	g
1	2	0	-1	3	k_2	$k_1 \cdot g$	$k_1 \cdot g^2$
1	1	0	-1	0	g^2	$k_2 \cdot g$	$k_2 \cdot g^2$

Therefore, we can derive a PIL program that validates the previous circuit as shown below:

```
include "config.pil";

namespace Plonk(%N);
pol constant QL, QR, QM, QO, QC;
pol constant SA, SB, SC;
pol constant L1;

pol commit a, b, c;

public pi = a(0);

// Public values check
L1 * (a - :pi) = 0;

// Plonk equation
pol ab = a*b;
QL*a + QR*b + QM*ab + QO*c + QC = 0;

// Copy-constraints check
{a, b, c} connect {SA, SB, SC};
```

1.10 Filling Polynomials

Until now we have only shown how to specify some kinds of constraints that several polynomials of a certain program described in PIL should satisfy to become *correct*. All these constraints, together with the constant polynomials inherent to the computation itself, specify the transition function underlying the program definition. In other words, changing either any of the constraints or the description of the constant polynomials produces a change in the program we are working on.

In this section, we are going to use Javascript and pilcom to generate a specific execution trace for a given PIL. To do so, we are going to compute a valid execution trace for the example of Section 1.6. As a remark, we will also use the library `pil-stark` whose utility is to provide a framework to setup, generate and verify proofs, to use a `FGL` class which mimics a finite field and it is required by some functions that provide the pilcom package.

First of all, under the scope of an asynchronous function called `execute`, we parse the provided PIL (which is, in our case, `main.pil`) into a Javascript object using the `compile` function of pilcom. In code we obtain:

```
const { FGL } = require("pil-stark");
const { compile } = require("pilcom");
const path = require("path");

async function execute() {
    const pil = await compile(FGL, path.join(__dirname, "main.pil"));
}
```

The pilcom package also provides two functions that use the `pil` object to create two crucial objects from it for the construction of the execution trace: the constant polynomials object and the committed polynomials object (using `newConstPolsArray` and `newCommitPolsArray` functions, respectively).

```
const { newConstantPolsArray, newCommitPolsArray, compile } = require("pilcom");

async function execute() {

    // ... Previous Code

    const constPols =  newConstantPolsArray(pil);
    const cmPols = newCommitPolsArray(pil);
}
```

Both such objects contain useful information about the PIL itself, such as the provided length of the program N, the total number of constant polynomials and the total number of committed polynomials. However, accessing these objects will allow us to fill the entire execution trace for that PIL. We can access a specific position of the execution trace using the syntax:

```
pols.Namespace.Polynomial[i]
```

being `pols` one of the previously introduced `constPols` and `cmPols` objects, `Namespace` being a specific namespace among the ones defined by the PIL files, `Polynomial` one of the polynomials defined under the scope of the previous namespace and i an integer in the set $[0, N-1]$ representing the row of the current polynomial. Using this we can now start to fill our polynomials.

In our example, we will use, as inputs for the trace, which are the ones introduced in the `Main.a` polynomial, an ascending chain of integers from 0 to 15 cyclically (because recall that we are only allowed to use 4 bits integers). We propose here two functions that fill the constant and committed polynomials accordingly.

```
async function buildConstantPolynomials(constPols, polDeg) {

    for (let i=0; i < polDeg; i++) {
        constPols.Global.BITS4[i] = BigInt(i & 0b1111);
        constPols.Global.L1[i] = i === 0 ? 1n : 0n;
        constPols.Negation.RESET[i] = (i % 4) == 3 ? 1n : 0n;
        constPols.Negation.FACTOR[i] = BigInt(1 << (i % 4));
        constPols.Negation.ISLAST[i] = i === polDeg-1 ? 1n : 0n;
    }
}
```

Figure 11: Generation of the constant polynomials.

```
async function buildcommittedPolynomials(cmPols, polDeg) {

    cmPols.Negation.a[-1] = 0n;
    cmPols.Negation.neg_a[-1] = 1n;

    for (let i=0; i < polDeg; i++) {

        let fourBitsInt = i % 16;

        cmPols.Main.a[i] = BigInt(fourBitsInt);
        cmPols.Main.neg_a[i] = BigInt(fourBitsInt ^ 0b1111);
        cmPols.Main.op[i] = FGL.mul(cmPols.Main.a[i], cmPols.Main.neg_a[i]);

        cmPols.Multiplier.freeIn1[i] = cmPols.Main.a[i];
        cmPols.Multiplier.freeIn2[i] = cmPols.Main.neg_a[i];
        cmPols.Multiplier.out[i] = cmPols.Main.op[i];

        let associatedInt = Math.floor(i/4);
        let bit = (associatedInt >> (i%4) & 1) % 16;
        cmPols.Negation.bits[i] = BigInt(bit);
        cmPols.Negation.nbits[i] = BigInt(bit ^ 1);


        let factor = BigInt(1 << (i % 4));
        let reset = (i % 4) == 0 ? 1n : 0n;
        cmPols.Negation.a[i] = factor*cmPols.Negation.bits[i]
        + (1n-reset)*cmPols.Negation.a[i-1];
        cmPols.Negation.neg_a[i] = factor*cmPols.Negation.nbits[i]
        + (1n-reset)*cmPols.Negation.neg_a[i-1];
    }
}
```

Figure 12: Generation of the committed polynomials.

Now that we have all the constant and committed polynomials filled in, we can check using a function called `verifyPil` that they indeed satisfy the constraints defined in the PIL file. We provide below the piece of code that construct the polynomials and check the constraints. If the verification procedure fails, we should not proceed to the proof generation because it will lead to false proof.

```
const { newConstantPolsArray, newCommitPolsArray, compile, verifyPil } = require("pilcom");

async function execute() {

    // ... Previous Code

    const N = constPols.Global.BITS4.length;

    await buildConstantPolynomials(constPols, N);
    await buildcommittedPolynomials(cmPols, N);

    const res =  await verifyPil(FGL, pil, cmPols , constPols);
    if (res.length != 0) {
        console.log("The execution trace do not satisfy PIL restrictions. Aborting...");
        for (let i=0; i<res.length; i++) {
            console.log(res[i]);
            return;
        }
    }
}
```

1.11 Generating a Proof Using `pil-stark`

Once the constant and the committed polynomials are filled, we can step to the proof generation stage. We can use the `pil-stark` Javascript package specially designed to work together with pilcom to generate STARK proofs about PIL statements. We will use three functions `starkSetup`, `starkGen` and `starkVerify` from the package. The first one is aiming for setting up the STARK, which is independent of the values of committed polynomials. This includes the computation of the tree of the evaluations of the constant polynomials. For executing the setup generation we ought to have an object called `starkStruct` which is specifying several FRI-related parameters such as the size of the trace domain (which must coincide with N, defined in PIL), the size of the extended domain (which together with the previous parameter specifies the correspondent *blowup factor*), the number of queries to be executed and the reduction factors for each of the FRI steps. We execute the setup using the code below:

```
const { FGL, starkSetup } = require("pil-stark");

async function execute() {

    // ... Previous Code

    const starkStruct = {
        "nBits": 10,
        "nBitsExt": 11,
        "nQueries": 128,
        "verificationHashType": "GL",
        "steps": [
        {"nBits": 11},
        {"nBits": 5},
        {"nBits": 3},
        {"nBits": 1}
        ]
    };

    const setup = await starkSetup(constPols, pil, starkStruct);
}
```

Now that we have set up the STARK, we can generate the proof using the `starkGen` function. We can do this task using the code below. Observe that the `setup` object contains inside a `starkInfo` field which contains, aside from all the `starkStruct` parameters, lots of useful information about the shape of the PIL itself.

```
const { FGL, starkSetup, starkGen } = require("pil-stark");

async function execute() {

    // ... Previous Code

    const resProof = await starkGen(cmPols, constPols, setup.constTree, setup.starkInfo);
}
```

Now that a proof has been generated we can be involved in the verification procedure invoking the `starkVerify` function. This function needs, as arguments, some information provided by the outputs of both the `starkSetup` and `starkGen` functions. If the output of the `starkVerify` function is `true`, the proof is valid. Otherwise, the verifier should invalidate the proof sent by the prover.

```
const { FGL, starkSetup, starkGen, starkVerify } = require("pil-stark");

async function execute() {

    // ... Previous Code

    const resVerify = await starkVerify(
    resP.proof, resP.publics, setup.constRoot, setup.starkInfo
    );

    if (resVerify === true) {
        console.log("The proof is VALID!");
    } else {
        console.log("INVALID proof!");
    }
}
```

References

[GW20] Ariel Gabizon and Zachary J. Williamson. plookup: A simplified polynomial protocol for lookup tables. Cryptology ePrint Archive, Report 2020/315, 2020. https://eprint.iacr.org/2020/315.

[GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Report 2019/953, 2019. https://eprint.iacr.org/2019/953.

[Her22a] Polygon Hermez. Pil compiler: Polynomial identity language (pil) compiler, 2022. https://github.com/0xPolygonHermez/pilcom.

[Her22b] Polygon Hermez. Pil stark: Generates a stark proof from a program written in pil language., 2022. https://github.com/0xPolygonHermez/pil-stark.

[PFM$^+$22] Luke Pearson, Joshua Fitzgerald, Héctor Masip, Marta Bellés-Muñoz, and Jose Luis Muñoz-Tapia. PlonKup: Reconciling PlonK with plookup. Cryptology ePrint Archive, Report 2022/086, 2022. https://eprint.iacr.org/2022/086.