tl;dr

AirScript is now much more expressive, a little bit faster, and one step close to composable STARKs. But there is still a lot of exciting work to do.

I've just released a new version of AirScript, and it is also now a part of genSTARK v0.6 library. A big part of AirScript code generator has been re-written to make it more modular. This will make implementing new language constructs easier. It also now produces a slightly more optimized JS code - which resulted in about 30% speed up in proving time (for moderately-complex computations). In the future, the same generator can be used to produce WASM code (instead of JS) which should improve proving time by another 2x - 3x.

The biggest change to AirScript in v0.5 is loop constructs. But before I get to loops, here is a list of other minor changes included in v0.5:

- Added support for vector slicing and element extraction. So, now you can do something like this: A[0..1]

and it will return a new vector consisting of the first 2 elements of vector A

. Element extraction works the same way as in all other languages: A[0]

returns a value at index 0 of the vector.

- Allowed direct references to register banks. So, now you can do: $r[0..1]

to get a vector which contains values of the first 2 state registers.

- Changed statement evaluation semantics to be more "Rust-like". Meaning, a block of statements always evaluates to the value of the last expression.
- Implemented unary operators for additive and multiplicative inverses. These can be applied to scalars, vectors, and matrixes.
- Enabled invoking transition function from transition constraints block. This eliminates writing a lot of redundant code for some type of STARKs.

To illustrate the power of new features, here is the code for defining transition function/constraints for Poseidon hash function (full working example is here):

transition 6 registers { // values to hash are passed in via input registers $i0-$i3 for each ($i0, $i1, $i2, $i3) {

```
    // initialize the execution trace
    init [$i0, $i1, $i2, $i3, 0, 0];

    // Poseidon hash transition logic
    for steps [1..4, 60..63] {
        // full rounds
        MDS # ($r + $k)^5;
    }

    for steps [5..59] {
        // partial rounds
        v5 <- ($r5 + $k5)^5;
        MDS # [...($r[0..4] + $k[0..4]), v5];
    }
}
```

}

enforce 6 constraints { for all steps { transition($r) = $n; } }

Notice how constraint definition just stipulates that transition($r) = $n;

. Also, #

is a matrix multiplication operator (for some reason, it shows up as a comment above).

Loops

AirScript now supports 2 loop constructs:

- for steps
- these are called segment loops.

- for each
- these are called input loops.

A detailed explanation of how these work is here. But in a nutshell: segment loops let you specify transition logic to execute at specific steps of the computation, while input loops let you specify how to consume inputs and move them into the execution trace.

Input loops can also be nested to create more complex STARKs. For example, here is AirScript transition function for verifying Merkle proofs. It uses 2 nested input loops:

transition 12 registers { for each ($i0, $i1, $i2, $i3) {

```
    // initialize state with first 2 node values
    init {
        S1 <- [$i0, $i1, $i2, $i3, 0, 0];
        S2 <- [$i2, $i3, $i0, $i1, 0, 0];
        [...S1, ...S2];
    }

    for each ($i2, $i3) {

        // for each node, figure out which value advances to the next cycle
        init {
            H <- $p0 ? $r[6..7] : $r[0..1];
            S1 <- [...H, $i2, $i3, 0, 0];
            S2 <- [$i2, $i3, ...H, 0, 0];
            [...S1, ...S2];
        }

        // execute Poseidon hash function computation for 63 steps
        for steps [1..4, 60..63] {
            S1 <- MDS # ($r[0..5] + $k)^alpha;
            S2 <- MDS # ($r[6..11] + $k)^alpha;
            [...S1, ...S2];
        }

        for steps [5..59] {
            S1 <- MDS # [...($r[0..4] + $k[0..4]), ($r5 + $k5)^alpha];
            S2 <- MDS # [...($r[6..10] + $k[0..4]), ($r11 + $k5)^alpha];
            [...S1, ...S2];
        }
    }
}

}
```

A detailed explanation of this example is here, but what it allows us to do is to specify that for every set of inputs in registers $i0

and $i1

there will be many sets of inputs in registers $i2

and $i3

. (I use sets 2 registers because in the example the registers can hold only 128-bit values - so, to hold a single 256-bit value, two registers are needed).

Future plans

There are still quite a few loose ends around loop constructs, and I'll keep working on making them more intuitive - but the biggest next thing is making AirScript composable. The idea here is that we'll be able to write AirScript for a hash function and then import it into a more complex AirScript etc. etc.

Other things I'd like to do (and would appreciate help from anyone), write AirScript for STARKs that can prove:

- Verification of updating sparse Merkle trees (adding/removing nodes).
- Verification of Schnorr signatures.