Anoma Research Topics | TECHNICAL REPORT

VampIR Bestiary

Joshua Fitzgerald[a] **and Alberto Centelles**[a]

[a]Heliax AG

* *E-Mail: joshua@heliax.dev, alberto@heliax.dev*

Abstract

This report delves into the comprehensive study of VAMP-IR Modules, detailing their transformations and comparisons. It explores specific aspects such as definitions, expressions, constraints, auxiliary data, and related algorithms. A thorough examination of IRs and their roles in Vamp-IR is conducted, complete with trade-offs and discussions on IRs resembling a proof system. The document illustrates the relationship among the presented IRs through diagrams, alongside an ideal pipeline. Existing and potential Vamp-IRs like Initial IR, Three-address code (3AC), Evaluated, Eliminated, Partially Evaluated and Partially Flattened (PEPF), Plonkish, R1CS, and AirScript are also discussed.

Keywords: VAMP-IR; Intermediate Representations; Three-address code (3AC); Rank-1 Constraint System (R1CS); AirScript;

(Received September 1, 2023; Published: January 15, 2024; Version: January 15, 2024)

Contents

1. Introduction

Intermediate representations (IRs) can be seen as a generalisation of arithmetic circuits. Popular IRs today are Rank-1 Constraint System (R1CS), Algebraic Intermediate Representation (AIR) and PlonkishAriel Gabizon (2021).

The current Vamp-IR implementation consists of:

1. a parser that returns an AST.
2. a typechecker that returns an initial IR that includes function calls.
3. a compiler that returns a final IR, a totally flattened IR called three-adress code (3AC).

The goal of this document is to analyse the structure, efficiency and other properties of different IRs that Vamp-IR could adopt in relation to different proving systems.

Since different proving systems may prefer different IRs over others, we will consider having multiple IRs within the Vamp-IR compiler and the costs of transforming one IR to another.

2. VAMP-IR Modules

Every IR in vamp-ir can be expressed as a *module*. A module is a tuple (pubs, definitions, expressions, auxiliary data). What differentiates one IR from another is the shape of the expressions used in the definitions and constraints and the presence and type of auxiliary data.

2.1. pubs. This is a list of variables that will be treated as *public*. These are normal variables in an arithmetic circuit, but may have designated locations in an IR.

2.2. definitions. A *definition* defines a function by binding a function body to a name, with designated inputs and, optionally, an output. Functions have scope and can contain other functions and constraints. The definitions also contain the instructions for *witness generation* and may include the `fresh` keyword and operations other than $+, -, \times$. Each variable ought to have a definition, and the definitions should be ordered such that each definition ought to refer only to previously defined variables.

2.3. expressions. An *expression* is a tree of infix operations $(+, -, \times)$ whose leaves are constants, variables, tuples, lists, or function calls.

2.4. constraints. A *constraint* is an equality between two expressions.

2.5. auxiliary data. Auxiliary data may be derived from an IR and included in the module to assist in efficient transformation to another IR. An example of auxiliary data that may be helpful is *registers* for transformations into STARK-like representations, e.g. AIR and AIRScript.

3. Transformations

3.1. Flattening. *Flattening* is the process by which constraints are broken down into a list of smaller constraints by adding auxiliary variables. Flattening increases the total number of constraints. For example, the constraint

$$d = a \times (b + c)$$

can be flattened to

$$v_1 = b + c$$
$$d = a \times v_1$$

by adding the auxiliary variable v_1.

3.2. Unflattening (elimination). *Unflattening* is the reverse of flattening, where auxiliary variables are eliminated and the total number of constraints decreases.

3.3. Partial flattening according to a list of allowed constraint forms. Total flattening and total unflattening are not difficult. However, most proof systems can only handle constraints of a certain size and shape, so totally unflattened constraints may not "fit" with a particular proof system. On the other hand, a totally flattened constraint system will not be very efficient, due to the large number of constraints which are only partially filled.

In "vanilla" Plonk, a maximally filled constraint will have the form

$$Q_m xy + Q_l x + Q_r y + Q_o z + Q_c + w = 0$$

where Q_m, Q_l, Q_r, Q_o, and Q_c are constants; x, y, and z are private variables; and w is a public input.

This same constraint expressed in 3AC, which is totally flattened, would be this list of 11 constraints:

$$v_1 = x \times y$$
$$v_2 = Q_m \times v_1$$
$$v_3 = Q_l \times x$$
$$v_4 = Q_r \times y$$
$$v_5 = Q_o \times z$$
$$v_6 = v_2 + v_3$$
$$v_7 = v_4 + v_5$$
$$v_8 = v_6 + v_7$$
$$v_9 = Q_c + w$$
$$v_{10} = v_8 + v_9$$
$$v_{10} = 0$$

Likewise, a single constraint in fully eliminated form may not fit into a single vanilla Plonk constraint, and will need to partially flattened into pieces that are small enough to fit in a single constraint.

In order to represent circuits efficiently we will need to use either a bottom-up or top-down process:

1. Partially unflatten 3AC constraints by eliminating variables until no more variables can be eliminated while remaining in the target form, or
2. Partially flatten totally eliminated constraints by adding auxililary variables until each constraint fits into the target form.

3.4. Algorithms for partial unflattening. Partial unflattening requires a function that can check if an expression is allowed in the target constraint system. Call this function *check*.

```
fn check (to_be_checked: Expression, reference_list: [Expression]) -> {0, 1}
```

1. Beginning in 3AC form, order the equations so that variables in the expressions on the right side refer occur on the left side of equations earlier in the list.
2. Create a hashmap with an entry for each equation, associating a variable to a 3AC expression. Ignore any simple equalities that use an already defined variable.

   ```
   H: Variable -> Expression
   ```

3. Loop through the list of equations eq_i

   (a) For *eq[i] = Equal(v1, InfixOp(v2, v3))*:

i. If *check(Subtract(InfixOp(H(v2), v3), v1, reference_list) = 1*
   A. Mutate the equation to *Equal(v1, InfixOp(H(v2), v3))*.
   B. Then if *check(Subtract(InfixOp(H(v2), H(v3)), v1, reference_list) = 1*, mutate *eq[i]* to *Equal(v1, InfixOp(H(v2), H(v3))*.

(b) Else, if *check(Subtract(InfixOp(v2, H(v3)), v1, reference_list) = 1*, mutate *eq[i]* to *Equal(v1, InfixOp(v2, H(v3)))*.

The function *check()* can be performed with subtree matching, with equivalence relations for commutativity, associativity, the distributive property, etc.

4. Comparison of IRs and their purposes in Vamp-IR

Each IR mentioned in this document has a purpose and each has differing qualities. Some are *basic* meaning they are not optimized for use in a particular proof system. The rest are considered *close to the proof system*.

4.1. Basic IRs.

- **3AC**: Three-address code (3AC) is a maximally flattened and completely evaluated IR. Because 3AC is completely evaluated its expressions contain only arithmetic constraints. Because it is maximally flattened its arithmetic constraints are "small" enough to fit in *any* constraint system in any proof system using arithmetic circuits. 3AC is therefore a truly universal arithmetic circuit representation and can easily be used in any proof system using arithmetic circuits. The drawback is that 3AC is very constraint-inefficient. Vamp-IR currently uses this IR for every backend.
- **Initial**: The Initial IR has not been flattened or evaluated at all and remains very close to the source language. This IR is not generally suitable for any backend. Vamp-IR currently uses this IR for type-checking.
- **Eliminated**: This IR has been completely evaluated, but maximally *un*flattened. The expressions in this IR are multivariate polynomials whose extraneous variables have been eliminated. This IR is perhaps best used for human inspection of circuits. Vamp-IR does not currently use this IR.
- **Evaluated**: This IR has been completely evaluated so its expressions are purely arithmetical. This IR could be useful in inspecting the arithmetical components of circuits before any flattening or unflattening optimizations have been applied which could obfuscate the circuit. Vamp-IR does not currently use this IR.
- **Partially Evaluated (PE)**: This IR has had some of its non-arithmetic expressions evaluated. The purpose of this IR is to target proof system backends with lookups, custom gates, or other optimized gadgets. The function calls that remain unevaluated should correspond to a list of available gadgets in the backend. Vamp-IR does not currently use this IR.
- **Partially Evaluated and Partially Flattened (PEPF)**: This IR has had some of its arithmetic constraints flattened (or unflattened) to be constraint-efficient in some backend implementation. Each arithmetic constraint in this IR should fit nicely into the constraint system of the backend. Vamp-IR does not currently use this IR.

4.2. Summary of trade-offs.

A **universal** IR means that there is a direct transformation from it to any other IR, (e.g. there are no ad-hoc rules such as auxiliary data, or unevaluated, backend-specific functions).

IR	Pros	Cons
3AC	▪ Immediately universal without further processing	▪ Constraint inefficient ▪ No custom gate or lookup information remains
Initial	▪ Useful for typechecking ▪ Human-readable ▪ No duplication of code ▪ Information that can be used to target custom gates or lookups remains	▪ Cannot be used directly in any backend without significant processing ▪ Not universal
Eliminated	▪ Universal after some flattening ▪ Closest to mathematical description of circuits found in specs	▪ Likely requires flattening before it can be used in any backend ▪ Unlikely to be constraint efficient ▪ No custom gate or lookup information remains
Evaluated	▪ Universal after some further processing ▪ Could be used to examine arithmetic components before flattening and unflattening	▪ Requires further processing before it can be used in a backend ▪ No custom gate or lookup information remains ▪ Unlikely to be constraint efficient
Partially Evaluated	▪ Lookup and custom gate information remains ▪ Can be very constraint efficient if partial evaluation targets backend gadgets	▪ Potentially universal but mappings to other IRs must be carefully constructed (see CCS paper Setty et al. (2023) for further details)
Partially Evaluated and Partially Flattened	▪ Most constraint efficient IR on this list if targeting backend gadgets ▪ Custom gate and lookup information remains ▪ No further processing needed for targeted backend	▪ Potentially universal but mappings to other IRs must be carefully constructed

4.3. IRs which are close to a proof system.

- **Plonkish**: For use in any Plonk-basedGabizon et al. (2019) proof system, including Halo2 ZCash Foundation (2023). Proof system implementations which are Plonk-based include:
  - zk-garage/plonk
  - zcash/halo2
  - aztecprotocol/barretenberg
  - noir-lang/acvm
- **AirScript**: For use in some STARK-based proof systems. Proof system implementations which use AirScript are:
  - facebook/winterfell

- **R1CS**: For use in the Groth16 Groth (2016), Marlin Chiesa et al. (2019) and Nova Kothapalli et al. (2021) proof systems. Proof system implementations using R1CS include:
  - zcash/bellman
  - scipr-lab/libsnark
  - arkworks-rs/marlin

5. IR Diagrams

5.1. Relationships of IRs. This diagram highlights the relationships between the different IRs explored in this document and its main transformations: flatten, evaluate and their inverses.

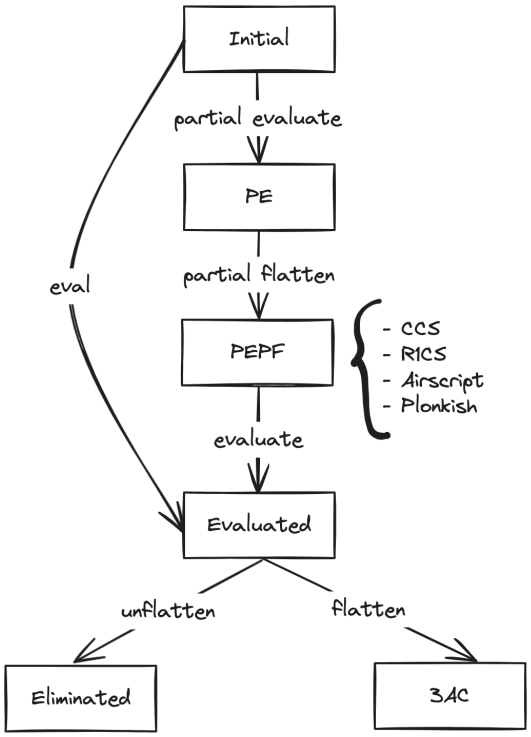

Fig. 1: Diagram of IR relationships

5.2. Ideal (simplified) pipeline. The most widely used IRs are neither fully evaluated nor fully flattened. For example, R1CS and Plonkish may be extended with lookups (unevaluated functions). R1CS has a simple form, but it is still not fully flattened, although its mapping to 3AC is linear. Plonkish has custom gates, thus requiring additional auxiliary data to generate its form.

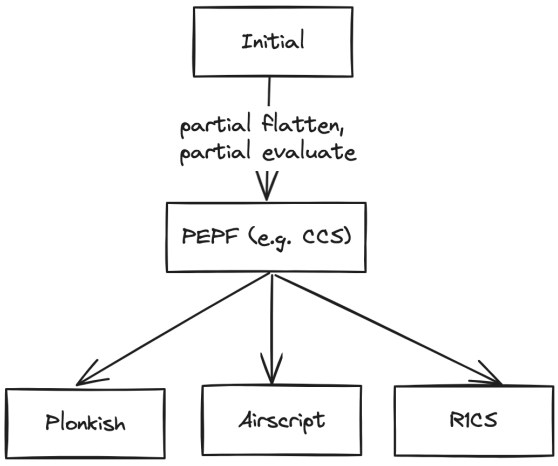

Fig. 2: Ideal (simplified) pipeline

6. Existing Vamp-IRs

This section analyses the two existing IRs in the current compiler:

1. Initial IR
2. Three-address code (3AC)

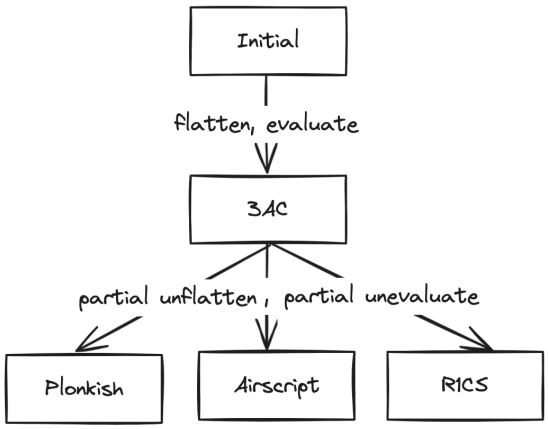

Fig. 3: Current pipeline

6.1. Initial IR. After the AST is type-checked we receive a module that retains function calls in its representation and has been only minimally flattened or processed. This IR is close to the initial AST from the parser. This IR has no auxiliary data.

The expression form is a tree whose nodes are labelled with $+$, -, or $\times$ and whose leaves are constants, variables, or function calls.

```
Constraint
    = Equal(Variable | Constant, Expr)
    | Equal(Variable | Constant, Function)
    | Equal((Variable | Constant, Variable | Constant, ...), Function)

Expr
    = Add(Expr, Expr)
    | Subtract(Expr, Expr)
    | Multiply(Expr, Expr)
    | Negate(Expr)
    | Variable
    | Constant
    | Function((Variable | Constant)^k)
```

6.1.1. Example.

```
pubs = [
    c
];

definitions = [
    def is_pyth x y z = {
        def r = (((x^2)+(y^2))-(z^2));
        def r_inv = fresh (1|r);
        (((r*r_inv)-1)*r) = 0;
        r*r_inv
    }
];

constraints = [
    new_score = score + is_pyth a b c;
];

auxiliary_data = [];
```

6.2. Three-address code (3AC). *3AC* is totally flattened and totally evaluated. The expression form is either a single node of a tree labelled with $+$, -, or $\times$, or is a leaf (i.e. a variable or a constant). Equations are a single, unique, variable on the left and an expression on the right. This IR has no auxiliary data.

6.2.1. Description. Allowed equations:

$$z = x + y$$
$$z = x - y$$
$$z = x \times y$$

... where x, y, and z may each be a variable or a constant.

```
Constraint
    = Equal(Variable | Constant, Expr)

Expr
    = Add(Expr, Variable | Constant)
    | Subtract(Variable | Constant, Variable | Constant)
    | Multiply(Variable | Constant, Variable | Constant)
    | Negate(Variable | Constant)
    | Variable
    | Constant
```

6.2.2. Example.

```
pubs = [
    c,
];

definitions = [
    v1 = a * a,
    v2 = b * b,
    v3 = v1 + v2,
    v4 = c * c,
    v5 = v3 - v4,
    v7 = 1|v5,
    v8 = v7,
    v9 = v5 * v8,
    v10 = v9 - 1,
    v11 = v10 * v5,
    v12 = v5 * v8,
    v13 = score + v12,
];

constraints = [
    v1 = a * a,
    v2 = b * b,
    v3 = v1 + v2,
    v4 = c * c,
    v5 = v3 - v4,
    v8 = v7,
    v9 = v8 * v5,
    v10 = v9 - 1,
    v11 = v10 * v5,
    v12 = v5 * v8,
    v11 = 0,
    new_score = v13,
];

auxiliary_data = [];
```

7. Potential Vamp-IRs

7.1. Evaluated. *Evaluated* is an IR with no remaining function calls, but its expressions may be in any state of flattening.

7.1.1. Description. Allowed equations:

$$y = \sum_{i=0}^{n} \left(c_i \prod_{j=0}^{m} x_j^{e_{i,j}} \right)$$

... where the c_i are constants, and y and the x_j are variables.

```
Constraint
    = Equal(Variable | Constant, Expr)

Expr
    = Add(Expr, Expr)
    | Subtract(Expr, Expr)
    | Multiply(Expr, Expr)
    | Negate(Expr)
    | Variable
    | Constant
```

7.1.2. Example.

```
pubs = [
    c
];

definitions = [
    def v1 = (((x^2)+(y^2))-(z^2)),
    def v2 = fresh (1|v1),
    def v3 = (v1*v2-1)*v1,
    def v4 = v1*v2,
];

constraints = [
    v1 = (((a^2)+(b^2))-(c^2));
    v3 = (v1*v2-1)*v1;
    v4 = v1*v2;
    new_score = score + v4;
    v3 = 0;
];

auxiliary_data = [];
```

7.2. Eliminated. *Eliminated* form is a maximally unflattened representation with no function calls remaining. All function calls have been evaluated and all unnecessary variables have been eliminated via substitution. The expression form is simply a multivariate polynomial with no restrictions on degree or number of variables. This IR has no auxiliary data.

Eliminated form is perhaps most useful for human inspection of circuits, say, in LaTeX.

7.2.1. Description. Allowed equations:

$$y = \sum_{i=0}^{n} \left(c_i \prod_{j=0}^{m} x_j^{e_{i,j}} \right)$$

... where the c_i are constants, and y and the x_j are variables.

```
Constraint
    = Equal(Variable | Constant, Expr)

Expr
    = Add(Expr, Expr)
    | Subtract(Expr, Expr)
    | Multiply(Expr, Expr)
    | Negate(Expr)
    | Variable
    | Constant
```

7.2.2. Example.

```
pubs = [
    c,
];

definitions = [
    v1 = fresh (1|((a*a + b*b) - (c*c))),
];

constraints = [
    0 = ((a*a + b*b) - (c*c))*v1 - 1) * ((a*a + b*b) - (c*c)),
    new_score = score + ((a*a + b*b) - (c*c))*v1,
];
```

7.3. Partially Evaluated and Partially Flattened (PEPF). This is similar to the Initial IR, except that calls to functions not on a "whitelist" have been evaluated, and the arithmetic expressions have undergone some flattening or unflattening. The whitelist forms the auxiliary data in this IR. Expressions retaining white-listed functions have the form:

`(tuple of outputs) = white_listed_fn(inputs)`

An expression not containing a white-listed function will be purely arithmetic with no function calls.

During synthesis, Vamp-IR will replace expressions retaining white-listed functions with an optimized gadget available in the backend.

7.3.1. Description. Allowed expressions:

$$y = \sum_{i=0}^{n} \left(c_i \prod_{j=0}^{m} x_j^{e_{i,j}} \right)$$

$$y = f(x1, x2, ...)$$

$$(y1, y2, ...) = f(x1, x2, ...)$$

```
Constraint
    = Equal((Variable | Constant)^n, Expr)

Expr
    = Add(Expr, Expr)
    | Subtract(Expr, Expr)
    | Multiply(Expr, Expr)
    | Negate(Expr)
    | Variable
    | Constant
    | Function((Variable | Constant)^k)
```

7.3.2. Example.

```
pubs = [
    c
];

definitions = [
    def is_pyth x y z = {
        def r = (((x^2)+(y^2))-(z^2));
        def r_inv = fresh (1|r);
        (((r*r_inv)-1)*r) = 0;
        r*r_inv
    },
];

constraints = [
    v1 = is_pyth a b c
    new_score = score + v1;
];
```

```
auxiliary_data = [
    "is_pyth"
];
```

7.4. Plonkish. *Plonkish* is a family of IRs parameterized by a configuration in its auxiliary data. The configuration in the auxiliary data consists of:

1. a list of k fixed columns (also called selectors)
2. a list of n advice columns (also called witnesses)
3. a list of m instance columns (also called public inputs)
4. D, the maximum constraint degree
5. a list of multivariate polynomial constraints (standard and custom gates)
6. a list of allowed lookup queries (partitions of advice columns)

See more at: https://zcash.github.io/halo2/concepts/arithmetization.html

Plonkish differs from AIR in that it doesn't require uniform circuits, and unlike R1CS, it can handle constraints of more than a degree of two. This versatility allows Plonkish to depict some program operations more succintly than either R1CS or AIR. However, proving with Plonkish might come at a greater cost per gate or constraint. For many relevant applications, the benefits from smaller circuits could surpass the costs of supporting more expressive gates.

7.4.1. Description. Allowed equations:

$$custom_{out} = custom(custom_{in})$$
$$lookup_{out} = lookup(lookup_{in})$$

...where the $n + m$ advice and instance columns are partitioned into a list of outputs ($custom_{out}$) and a list of inputs ($custom_{in}$) for each custom gate. For lookups, $lookup_{out}$ and $lookup_{in}$ only partition the n advice columns.

Custom gates are multivariate polynomials that must be satisfied by the $k + n + m$ available advice and instance columns, along with a partition of the advice and instance columns into "inputs" and "outputs".

Lookup queries are a partition of the n advice columns into "inputs" and "outputs", much like a function signature. The actual logic of the function is defined elsewhere.

7.4.2. Example. An example of a circuit using the configuration defined in the auxiliary data. The auxiliary data is given in JSON and defines a configuration with width-5 constraints with a custom gate (`fifth_power`) and a lookup gate (`matrix`).

Notice that some definitions have been written to correspond to the available custom gates and lookups in the configuration. A programmer desiring efficient circuits would write their Vamp-IR code in this manner, but this is not enforced.

This is a 16-bit hash circuit inspired (extremely loosely) by the Poseidon SNARK friendly hash function which uses matrix multiplication and fifth powers.

```
pubs = [];

definitions = [
    def fifth_power x1 = {
        def x2 = x1^5;
        x2
    },
    def matrix x1 x2 = {
        def new_1 = 3*x1 + 7*x2;
        def new_2 = -2*x1 + 5*x2;
        def x3 = range_8 (fresh (new_1 % 2^8));
        def x4 = range_8 (fresh (new_2 % 2^8));
        (x3, x4)
    },
    def bool x = {
        x^2 = x;
        x
    },
    def range_2 x = {
```

```
        def lo = bool (fresh (x % 2));
        def hi = bool (fresh (x \ 2));
        x = 2*hi + lo;
        x
    },
    def range_4 x = {
        def lo = range_2 (fresh (x % 2^2));
        def hi = range_2 (fresh (x \ 2^2));
        x = 2^2*hi + lo;
        x
    },
    def range_8 x = {
        def lo = range_4 (fresh (x % 2^4));
        def hi = range_4 (fresh (x \ 2^4));
        x = 2^4*hi + lo;
        x
    },
    def hash_round lo hi = {
        matrix (fifth_power lo) (fifth_power hi)
    },
    def hash x = {
        def lo = range_8 (fresh (x % 2^8));
        def hi = range_8 (fresh (x \ 2^8));
        def (new_lo, new_hi) = iter 10 hash_round lo hi;
        2^8*hi_mod + lo_mod
    },
];

constraints = [
    y = hash x,
];

auxiliary_data = {
    "fixed_columns": ["a", "b", "c", "d", "e"],
    "advice_columns": ["x1", "x2", "x3", "x4"],
    "instance_columns": [],
    "D": 5,
    "custom_gates": [
        {
            "standard": {
                "constraint": a*x1*x2 + b*x1 + c*x2 + d*x3 + e*x4",
                "inputs": ["x1", "x2", "x4"],
                "outputs": ["x3"]
            },
            "fifth_power": {
                "constraint": "x2 - x1^5",
                "inputs": ["x1"],
                "outputs" ["x2"]
            },
        }
    ],
    "lookups": [
        {
            "matrix": {
                "inputs": ["x1", "x2"],
                "outputs": ["x3", "x4"]
            }
        }
    ]
}
```

7.5. R1CS. *R1CS* uses constraints of the form $C = A \times B$ where A, B, and C are *linear combinations* of variables and constants. For convenience, we designate one variable in C to be the "out" variable and write only it on the left.

The rest of the terms in C are moved to the right. This IR needs no auxiliary data.

7.5.1. Formal definition. Allowed equations:

$$z = \left(\sum_{i=0}^{n} a_i x_i\right)\left(\sum_{i=0}^{n} b_i x_i\right) - \left(\sum_{i=0}^{n} c_i x_i\right)$$

7.5.2. Example.

```
pubs = [
    c,
];

definitions = [
    v1 = (2*a)*b,
    v2 = (a + b - c)*(a + b + c) - v1,
    v3 = 1|v2,
    v4 = v2 * v3,
];

constraints = [
    v1 = (2*a)*b,
    v2 = (a + b - c)*(a + b + c) - v1,
    v4 = v2 * v3,
    0 = (v4 - 1)*v2,
    new_score = v4 + score,
];

auxiliary_data = [];
```

7.6. AirScript. *AirScript* is an intermediate representation used by PolygonMiden which compiles to the Winterfell STARK proof system. STARKs are very inefficient unless the circuit conforms to certain structure: namely, that the entire circuit can be written as a bounded loop of a relatively small subcircuit.

If a Vamp-IR circuit has this structure it is fairly simple to compile it into AirScript by computing some auxiliary data.

7.6.1. Auxiliary Data Format. In order to compile Vamp-IR to AirScript we need:

1. a list of private main register names
2. (optional) a list of private auxiliary register names
3. at least one public register with a name and a length
4. a hashmap from Vamp-IR variable names to AirScript register names
5. every variable in the Vamp-IR document needs to be mapped uniquely to a private or public register name with the appropriate suffix if needed: .first, .last, ')
6. a partition of the constraints into boundary constraints and integrity constraints

7.6.2. Generating the Auxiliary Data. Consider the simple case of an iterated function in Vamp-IR. The function below has the same number of inputs as the outputs and so it works very much like a register machine with registers *a*, *b*, *c*, and *d*. The entire circuit is a single function iterated 8 times, along with some constraints on the initial inputs and final outputs of the function.

```
def my_function (a, b, c, d) = {
  def a_next = a + c;
  def b_next = b + d;
  def c_next = c - 1;
  def d_next = 2*d;

  (a_next, b_next, c_next, d_next)
};

(a_out, b_out, c_out, d_out) = iter 8 my_function(a_in, b_in, c_in, d_in);

c_out = 0;
b_in = a_in;
```

1. Evaluate all function calls within *my_function*, replacing them with the appropriate constraints and definitions. Do *not* evaluate the intrinsic *iter*.

2. Condense the constraints within *my_function* by eliminating all intermediate variables. This will reduce the number of registers we need.

3. Create a register name as a string for each non-public parameter of *my_function* and each internal non-public variable in *my_function*, *excluding the output variable(s)*. Add an entry for each variable to the variable name map.

4. Create a register name for each public input accessed by *my_function*. Pubs are declared globally by name in Vamp-IR so these names should be reused here. A list of these names and the length of each register needed becomes the data in *public_inputs* above. Add an entry for each public variable to the variable name map. AirScript requires at least one public variable, so if the list of pubs is empty a dummy public variable should be created.

5. Loop through the list of expressions and add an entry to the partition hashmap for each expression. Boundary constraints will only reference variables mapped to *(register name).first* or *(register name).last*. Integrity constraints will only reference variables mapped to *(register name)* or *(register name)'*. A constraint that references variables from both classes is invalid.

7.6.3. Example.

```
pubs = [];

definitions = [
    def my_function (a, b, c, d) = {
      def a_next = a + c;
      def b_next = b + d;
      def c_next = c - 1;
      def d_next = 2*d;

      (a_next, b_next, c_next, d_next)
    },
];

constraints = [
    a_next = a + c,
    b_next = b + d,
    c_next = c - 1,
    d_next = 2*d,
    c_out = 0,
    b_in = a_in,
];

auxiliary_data = {
    "main_registers": ["a", "b", "c", "d"],
    "auxregisters": [],
    "public_inputs": ["dummy", 1],
    "variable_mapping": {
        "a": "a",
        "b": "b",
        "c": "c",
        "d": "d",
        "a_in": "a.first",
        "b_in": "b.first",
        "c_in": "c.first",
        "d_in": "d.first",
        "a_out": "a.last",
        "b_out": "b.last",
        "c_out": "c.last",
        "d_out": "d.last",
        "a_next": "a'",
        "b_next": "b'",
        "c_next": "c'",
```

```
        "d_next": "d'",
    },
    "periodic_columns": [],
    "constraints": {
        "integrity": [
            "a_next = a + c",
            "b_next = b + d",
            "c_next = c - 1",
            "d_next = 2*d",
        ],
        "boundary": [
            "c_out = 0",
            "b_in = a_in",
        ]
    }
}
```

7.6.4. Compiling more complex Vamp-IR circuits to AirScript. The scheme above does not make use of *periodic columns*. Periodic columns allow different constraints to be active on different cycles. We can use the periodic columns to allow for more complex Vamp-IR circuits which have some constraints on computation done before or after the iterated function. (Hash functions come to mind as an example of this.)

The scheme above also does not use any *auxiliary trace columns*. Auxiliary trace columns can access new randomness from the Verifier. (Author note: I am not sure what auxiliary trace columns are used for. I suspect they are needed for folding or recursion.)

7.7. CCS. *Custom constraint system* or CCS is a new constraint system that generalizes Plonkish, R1CS, and AIR constraints. CCS is more general in the sense there are costless reductions from instances of these IRs to equivalent CCS instances.

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

$$\sum_{i=0}^{q-1} c_i \cdot \circ_{j \in S_i} M_j \cdot z = \mathbf{0} \quad [1]$$

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

$M_j \cdot z$ denotes matrix-vector multiplication. $\circ$ denotes the Hadamard 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:

$$c_0 \cdot \overbrace{(M_{j_0} \cdot z \circ ... \circ M_{j_{|S_0|-1}} \cdot z)}^{j_i \in S_0} + ... + c_{q-1} \cdot \overbrace{(M_{j_0} \cdot z \circ ... \circ M_{j_{|S_{q-1}|-1}} \cdot z)}^{j_i \in S_{q-1}} = \mathbf{0}$$

Definition 1 (CCS). *A CCS structure S consists of:*

1. *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. (You can think of these matrices as a generalisation of the A, B, C matrices in R1CS that encode the constraints).*
2. *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. (You can think of S_i as containing the pointers to the matrices M_i for each of the addends in the CCS equation).*
3. *a sequence of q constants $[c_0, ..., c_{q-1}]$, where each constant is from $\mathbb{F}$*
4. *size bounds $m, n, N, l, t, q, d \in N$ where $n \geq l$*
    - *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*

7.7.1. Lookups. The definition of CCS with lookups (CCS+) is exactly as the definition of CCS, with the following two properties

- a lookup table T , where T is a set of values from $\mathbb{F}$
- a sequence of lookup operations L, where each entry in L is in the range $[n]$.

and a CCS+ structure-instance tuple (S, x) is satisfied by a CCS+ witness w if w satisfies it as a CCS structure-instance tuple and for each lookup operation o in L, $z[o] \in T$.

7.7.2. 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 $t = 3, q = 2, d = 2, M_0 = A, M_1 = B, M_2 = C, S_1 = \{0, 1\}, S_2 = \{2\}, c_0 = 1, c_1 = -1$.

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

7.7.3. Formal definition. Allowed equations:

$$z = \left(\sum_{i=0}^{n} a_i x_i\right)\left(\sum_{i=0}^{n} b_i x_i\right) - \left(\sum_{i=0}^{n} c_i x_i\right)$$
$$lookup_{out} = lookup(lookup_{in})$$

7.7.4. Example code.

```
pubs = [
    c,
];

definitions = [
    v1 = (2*a)*b,
    v2 = (a + b - c)*(a + b + c) - v1,
    v3 = inverse v2,
    v4 = v2 * v3,
];

constraints = [
    v1 = (2*a)*b,
    v2 = (a + b - c)*(a + b + c) - v1,
    v4 = v2 * v3,
    0 = (v4 - 1)*v2,
    new_score = v4 + score,
];

auxiliary_data = {
    "lookups": [
        {
            "inverse": {
                "inputs": ["x1"],
                "outputs": ["x2"]
            }
        }
    ]
};
```

7.7.5. Representing Plonkish in CCS. Plonkish circuits are typically described in terms of gate checks, copy checks, permutation checks, etc. and the Plonk proving system is a set of gadgets to prove these types of checks. That is, description of Plonkish re typically closely tied to how the Plonk proof system internally works.

Definition 2 (Plonkish). *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*

$$g(X_1, ..., X_t) = \sum_{i=0}^{q} k_i \cdot X_1^{i_1} ... X_t^{i_t}$$

where $i_1 + ... + i_t \leq d$ for all i.

- *a vector of constants called 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 $\{0, ..., n+e-1\}$.*

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

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*

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

$$g(z[T_i[1]], ..., z[T_i[t]]) = 0 \quad [2]$$

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

How does the Plonkish representation that CCS proposes compare to the existing representation? Plonkish is often specified with a combination of gate constraints and copy constraints. Plonkish in CCS eschews copy constraints by using a "deduplicated" version of the satisfying assignment whose length is shorter than a Plonkish satisfying assignment by the number of copy constraints.

7.7.6. CCS to Plonkish. 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 $M_0, ..., M_{t-1}$? There is a row for each constraint in $S_{Plonkish}$, 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.

7.7.7. Considerations on Vamp-IR. At first glance, we considered R1CS as a good IR candidate, since its transformation to Plonkish is quite straightforward, while the inverse transformation is not so easy. CCS seemed too general to be useful as a *fixed* IR. However, it doesn't have to be fixed.

We propose CCS with lookups (CSS+) as an IR that is parameterised by the user, that is, the user passes the configuration parameters $(m, n, N, l, t, q, d, [M_0, ..., M_{t-1}], [S_0, ..., S_{q-1}], [c_0, ..., c_{q-1}])$ from S_{CCS}. These configuration parameters become part of the auxiliary data. As we saw, we can represent R1CS as CCS by setting $t = 3, q = 2, d = 2, M_0 = A, M_1 = B, M_2 = C, S_1 = \{0, 1\}, S_2 = \{2\}, c_0 = 1, c_1 = -1$. We also saw that CCS can represent different IRs such as Plonkish or Airscript by setting certain configuration parameters.

How does it affect the structure of the code? We want to provide a default configuration and allow the user to change that configuration before compiling. This configuration will be auxiliary data that is carried along the compilation process.

Given the seemingly efficient transformations among CCS configurations, we could take a circuit and transform it into different configurations to figure out the best.

7.7.8. Additional thoughts. Currently, the Vamp-IR compiler only has two IRs and transformations such as evaluation or flattening happen at the same time. For certain backends such as Halo2, some gadgets benefit from *partial* evaluation. On the other hand, a totally flattened circuit such as 3AC is never the most efficient (e.g. circuit size is bigger than in other forms), although in this form, many compiler optimisations can be applied.

References

Ariel Gabizon. Plonkish Arithmetization, 2021. URL https://hackmd.io/@aztec-network/plonk-arithmetiization-air. (cit. on p. 2.)

Srinath Setty, Justin Thaler, and Riad Wahby. Customizable constraint systems for succinct arguments. Cryptology ePrint Archive, Paper 2023/552, 2023. URL https://eprint.iacr.org/2023/552. https://eprint.iacr.org/2023/552. (cit. on p. 5.)

Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Paper 2019/953, 2019. URL https://eprint.iacr.org/2019/953. https://eprint.iacr.org/2019/953. (cit. on p. 5.)

ZCash Foundation. Halo2, 2023. URL https://github.com/zcash/halo2. (cit. on p. 5.)

Jens Groth. On the size of pairing-based non-interactive arguments. Cryptology ePrint Archive, Paper 2016/260, 2016. URL https://eprint.iacr.org/2016/260. https://eprint.iacr.org/2016/260. (cit. on p. 6.)

Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Psi Vesely, and Nicholas Ward. Marlin: Preprocessing zksnarks with universal and updatable srs. Cryptology ePrint Archive, Paper 2019/1047, 2019. URL https://eprint.iacr.org/2019/1047. https://eprint.iacr.org/2019/1047. (cit. on p. 6.)

Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. Nova: Recursive zero-knowledge arguments from folding schemes. Cryptology ePrint Archive, Paper 2021/370, 2021. URL https://eprint.iacr.org/2021/370. https://eprint.iacr.org/2021/370. (cit. on p. 6.)