Wizkit Candidate IR Proposal

Table of Contents

1. Introduction

During the April 2022 SIEVE Principal Investigator's Meeting, the Wizkit team outlined a plan for a high level IR (IR2 or Translation-IR) with semantics defined by a translation to a lower level flat circuit IR (IR0 or Circuit-IR). This design addresses a key challenge facing the program: the conflict between our mandate to come up with a common IR and our tendency to tightly integrate each ZK backend with a specifically tailored IR. Further complicating this issue is the wide variety in design choices and capabilities of each backend-specific IR. The SIEVE program has seen everything from C++ libraries through R1CS as viable backend-specific IRs. With this in mind, Wizkit's IR proposal begins with TA1 teams producing statements in the Translation-IR, and TA2s taking one of the following approaches to prove the statement.

1. Interpret the Translation-IR statement, and emit a directive to a lower-level IR at each expression. This is the approach taken in specifying the Translation-IR, using the Circuit-IR as a target. We expect that an R1CS translation would also be easy to implement, as would be small deviations to the Circuit-IR.
2. Perform a syntax to syntax translation from the Translation-IR to a similarly capable, but backend-specific IR.
3. Interpret the Translation-IR statement, and directly evaluate each expression in ZK, rather than emitting a lower-level IR.
4. A hybrid approach utilizing two or more of the previous approaches.

This overview will start by outlining some common elements of both IRs. Then it will describe the canonical forms for the Circuit-IR and the Translation-IR. Finally, it will outline scenarios where deviations from the canonical forms are permitted or encouraged.

1.1. Multi Field Circuits

To most practitioners of ZK, a single prime field is chosen at the beginning of a proof and used throughout. However, for some applications it is desirable to use multiple primes for different elements within a single larger proof. For example a large and expensive prime may be needed to verify public-key signatures, while a medium sized prime is necessary for large scale business logic.

To accommodate these applications, the IR must allow for multiple fields within a single relation. To TA1 the field must describe the type of a wire, while to TA2 these wires actually belong to multiple independent proofs. An analogy to the real world might be a circuit card with transistor logic on one side and high voltage on the other.

Occasionally information from one field will be required in another. The IR models this using a conversion gate with inputs in one field and outputs in another. To continue the analogy, a relay would allow information to flow from transistor logic into high voltage, or in reverse, an analog-digital converter. In ZK, methodologies must be developed and used to show equivalence of inputs and outputs across independent proofs or even across different proof systems.

1.2. Feature Categorization Framework

To aide in designing the IR, the following decision framework is used to categorize features as belonging to the IR's core functionality or to a library or a plugin. IR features can be categorized in two aspects. First, a feature may be self-contained, or it may have idiosyncrasies which cause it to interact poorly with other features (for lack of a better term, we'll call these "control features"). For example, ring-based ZK would be simple to isolate into its own wiring space and to share `@add` and `@mul` gate syntax with field-based ZK. By contrast, IR1 introduced a syntax for private condition switch statements which necessitated changes to the semantics of `@short_witness` stream consumption — since the active branch was chosen after all branches were executed, the stream was rewound before executing each case. Second, a feature may be necessary to enable a class of computation or optimization. For example, a dot product can be simulated using combinations of `@mul` and `@add`. However, conversion gates are necessary to enable comparison of values in different fields, and this functionality cannot be simulated with combinations of simple gates.

Requirements of the Circuit-IR

The Circuit-IR's purpose is to serve as the translation target for the Translation-IR. As such, it must first be capable of representing any circuit which is desirable to emit from the Translation-IR. Secondly, it should be simple and easy for any backend to parse it and prove whether or not the relation is valid. Lastly, it should be flexible enough that self-contained features may be optimized with plugins. What the Circuit-IR likely will not be is succinct or compact; rather, circuits are expected to be enormous. Further, it is expectable that although any backend can handle the Circuit-IR, specific samples of the Circuit-IR are likely to be tailored for a particular backend through the use of preferred primes and plugins.

Requirements of the Translation-IR

The Translation-IR will make up for the Circuit-IR's deficiencies with increased complexity. As a program, control features will act in harmony to both shrink relation sizes and increase flexibility. With increased flexibility, interoperability will be restored for specific samples of the Translation-IR. For example, primes might be left unspecified in the Circuit-IR, or even substituted during compilation. With increased semantic information — a type system and control flow — the Translation-IR will also enable greater opportunities for optimization. What the Translation-IR will not enable is private condition control flow.

2. Motivating Examples

Before outlining the specifics of the IR, we have a few examples which we think help illustrate the motivation and design decisions which went into this proposal.

The examples shown in this section predate the existence of any execution environment and are as of yet untested.

2.1. Matrix Multiplier

The matrix multiplier demonstrates a size- and field-agnostic functionality, vectorized computation, and function composition. Evaluating the Translation-IR occurs in essentially two phases; a public translation phase evaluates publicly known variables and expressions, and emits private expressions as directives in a circuit. Public variables such as `field F` and `size s` are known during the public phase and may be used to define the type of variables emitted to the output circuit. Variables with `wire` in their type only have values during the circuit phase, so operations on these variables use gates such as `@mul` which are emitted to the output.

First up, the `vector_mul` function performs a pairwise multiplication across two private vectors. The type `wire F[s]` denotes a vector of `s` many wires using the field `F`. In the body, a for loop traverses across the input and output vectors, and the `@mul` gate emits a directive on each iteration of the loop.

```
@function vector_mul(@out: wire F[s] os, @in: field F, size s, wire F[s] ls, wire F[s]
rs)
  o in os <- for i (l in ls, r in rs)
    o <- @mul(l, r);
  end
@end
```

The `vector_sum` function will add all the elements of an input list to create a single output wire. This loop adds each value in a list to a temporary variable. Because the `@add` actually emits a directive, a new wire is created on each iteration of the loop and discarded in the next. This example also introduces some array indexing: first to take the `0`th element, and then to take a subset of the array starting at item `1` through the end of the list (`s - 1`).

```
@function vector_sum(@out: wire F o, @in: field F, size s, wire F[s] ls)
  tmp <- ls[0];
  for i (l in ls[1 ... (s - 1)]) modifies tmp
    tmp <- @add(tmp, l);
  end
  o <- tmp;
@end
```

The `vector_dotprod` function simply combines the prior two functions, first multiplying out a new vector and then summing its elements.

```
@function vector_dotprod(@out: wire F o, @in: field F, size s, wire F[s] ls, wire F[s]
rs)
  tmp <- @call(vector_mul, F, s, ls, rs);
  o <- @call(sum, tmp);
@end
```

Lastly, the `matrix_mul` function accepts three size parameters to define two matrices sized such that they may be multiplied together. It uses a pair of for loops to repeatedly invoke the dot product on a row of the `L` matrix and a column of the `R` matrix. It uses an index expression to select a row and a special "dimension reordering" index expression to take a column of the matrix.

In the column expression, dimension reordering would ordinarily produce a vector in the output with gaps between the elements. This can cause problems in the Circuit-IR where contiguity requirements may forbid gaps between elements in a list of wires. However, the `@wire` expression (among other things) will, when necessary, emit copy directives to the output creating a contiguous range of wires.

```
@function matrix_mul(@out: wire F[a, c] O, @in: field F, size a, size b, size c, wire
F[a, b] L, wire F[b, c] R)
  row in O <- for i (l in L)
    o in row <- for j (r in R[^1, ^0])
      o <- @call(vector_dotprod, l, @wire(r))
    end
  end
@end
```

When translating these functions to a circuit, there are a few potential approaches to the translation. Even with strict adherence to this proposal’s defined semantics, function inlining could emit either a massive `matrix_mul` function body with repeated sequences of `@mul` and `@add` or a more compact body with just repeated calls to `vector_dotprod`. If plugins are enabled, the body of any or all of these functions may be replaced with a single plugin directive indicating that the backend must recognize the functionality by a predefined name. Alternate translation approaches — such as translation to a C library -- might emit a preexisting function call in place of a plugin or simply emit C code for this same code flow.

2.2. Arithmetic Multiplexer Function

The arithmetic multiplexer demonstrates control flow using publicly known integers. It is also the basis for simulating private control flow in ZK, where all branches must be evaluated before selecting results from the "correct" branch. This example uses Fermat’s Little Theorem to create a select bit for every index in an array. Every bit should have the value `0` except for the active branch which has `1`. Then the input array and the select bit array may be used in a dot product to produce the single output value.

Fermat’s Little Theorem allows us to raise a number to the power of *prime-1* and we get either 0 if the number was 0 or 1 if it were non-zero (modulo *prime*). This means we need the recursive fast-power algorithm. Since *prime-1* is known ahead of time, we can use public arithmetic to condition our recursion. Thus, our exponent function has a publicly known field (prime), a publicly known exponent, and a private base.

```
@function exponent(@out: wire F result, @in: field F, integer exp, wire F base)
  result <- if (exp == 1)
    result <- base;
  elif ((exp % 2) == 0)
    tmp <- @call(exponent, F, (exp / 2), base);
    result <- @mul(tmp, tmp);
  else
    tmp <- @call(exponent, F, exp - 1, base);
    result <- @mul(tmp, base);
  end
@end
```

When translating the exponent function, the translator should probably inline recursive calls. In the field *GF(7)*, the exponent would start as *6* and produce the following Circuit-IR function. Notice that in the Circuit-IR variables are numbered within each field's numbering space. So in this example `0` refers to the numbering space for the field *GF(7)* and `$n` refers to a numbered wire within that space.

Circuit IR

```
@function(exponent_7_6, @out: 0:1, @in: 0:1)
  // $0 output wire
  // $1 input wire
  $2 <- @mul(0, $1, $1) // stack level 3, exp is 2
  $3 <- @mul(0, $2, $1) // stack level 2, exp is 3
  $0 <- @mul(0, $3, $3) // stack level 1, exp is 6
@end
```

The multiplex function is then simple to write. First a loop generates a `selects` vector with length `s`, then a call to `vector_dotprod` calculates the selected output.

```
@function arithmetic_multiplex(@out: wire F o, @in: field F, size s, wire F branch,
wire F[s] candidates)
  F_minus_1 <- (@integer(F) - 1);
  select in wire F[s] selects <- for i
    tmp <- @addc(branch, (F_minus_1 - @integer(i)));
    exp <- @call(exponent, F, F_minus_1, tmp);
    select <- @addc(@mulc(exp, F_minus_1), 1);
  end

  o <- @call(vector_dot, F, s, candidates, selects);
@end
```

2.3. Private Index Arrays or Random Access Memory (RAM)

While the IR cannot directly support RAM, it can simulate some of its behavior using circuits. Although this won't enable the performance and scalability of true RAM, it does allow for the IR to provide an interface for RAM. Backends with support for RAM may use the IR's interface to interoperate their RAM with a variety of frontends. Backends lacking support for RAM are still enabled to perform proofs at small scales until the naive RAM simulation's poor scalability becomes prohibitive. This is not unlike the prior matrix multiplication example where schoolbook multiplication is used, even though certain backends may scale bigger and faster using bespoke algorithms for ZK.

Using Translation-IR structs, an interface and a naive implementation for RAM can be developed.

```
struct RAM
  field F;
  size s;
  wire F[s] buffer;

  // Create a RAM with all elements assigned 0
  @function RAM_create(@out: struct RAM r, @in: field G, size t)
    b in wire G[t] buf <- for i
      b <- @wire(F: 0);
    end

    // The special create directive will instantiate a struct
    r <- create(F: G, s: t, buffer: buf);
  @end

  // Read one element from the RAM
  @function RAM_read(@out: wire G o, @in: field G, struct RAM r, wire G idx)
    // The special access directive allows element access to a struct
    f, s, buf <- access(r: F, s, buffer);

    // Call out to an arithmetic multiplexer here to behave like RAM.
    // This statement also indirectly checks that f and G are the same field
    // because it requires o and idx to have both field f and G
    o <- @call(arithmetic_multiplex, f, s, idx, buf);
  @end

  // Write one element to the RAM
  @function RAM_write(@in: field G, modify struct RAM r, wire G idx, wire G val)
    f, s, buf <- access(r: F, s, buffer);

    f_minus_1 <- (@integer(f) - 1);

    n in nbuf <- for i (b in buf)
      // create a selector bit and an opposite selector bit
      tmp <- @addc(idx, (f_minus_1 - @integer(i)));
      o_sel <- @call(exponent, f, f_minus_1, tmp);
      n_sel <- @addc(@mulc(o_sel, F_minus_1, 1), 1);

      // select either the new value or the old value
      n <- @add(@mul(val, n_sel), @mul(b, o_sel));
    end

    // modify an annotated input struct
    r <- @modify(buffer : nbuf);
  @end
end
```

The ability to use a RAM-like interface enables the IR to encode algorithms such as merge sort which require only RAM and public control flow. Merge sort can be implemented using just RAM and comparisons.

```
// o is 1 if l < r, and 0 otherwise
```

```
@function less_than(@out: wire F o, @in: field F, wire F l, wire F r)
  /* Omitted */
@end

@function merge_sort(@in: field F, struct RAM ram, size data_start, size
scratch_start, size length)
  for i repeat length
    tmp <- @call(RAM_read, F, ram, @wire(F: (i + data_start)));
    @call(RAM_write, F, ram, @wire(F: (i + scratch_start)), tmp);
  end

  @call(merge_sort_helper, F, ram, data_start, scratch_start, length);
end

@function merge_sort_helper(@in: field F, struct RAM ram, size data_start, size
scratch_start, size length)
  if (length > 1)
    // recurse over left and right half
    mid = (length / 2);
    @call(merge_sort, F, ram, scratch_start, data_start, mid);
    @call(merge_sort, F, ram (scratch_start + mid), (data_start + mid), mid);

    // then merge
    a <- @wire(F: scratch_start);
    b <- @wire(F: scratch_start + length);
    c <- @wire(F: data_start);
    for i repeat length modifies a, b, c
      // Read from RAM and then compare
      a_val <- @call(RAM_read, F, ram, a);
      b_val <- @call(RAM_read, F, ram, b);
      cond <- @call(less_than, F, a_val, b_val);
      anticond <- @addc(@mulc(cond, (integer(F) - 1)), 1);

      // Select a value to write to RAM
      c_val <- @add(@mul(a_val, cond), @mul(b_val, anticond));
      @call(RAM_write, F, ram, c, c_val);

      // update the indexes
      a <- @add(@mul(@addc(a, 1), cond), @mul(a, anticond));
      b <- @add(@mul(@addc(b, 1), anticond), @mul(b, cond));
      c <- @addc(c, 1);
    end
  end
@end
```

RAM also opens up the possibility of encoding a CPU in the IR, although this is not shown.

3. Common Elements

3.1. IR Header

The IR header starts with a version number and a resource type indicator. The current version number is `2.0.0-wtk-candidate`. The type tells how to parse and use the remainder of the IR resource. Here is an example header with the `circuit` type.

```
version 2.0.0-wtk-candidate;
circuit;
```

These are the available resource type indicators.

`circuit`
: a relation in the Circuit-IR.

`translation`
: a relation in the Translation-IR.

`library`
: a library which may be included by a Translation-IR relation.

`instance`
: a stream of publicly known variables.

`short_witness`
: a stream of prover-only variables.

3.2. Field and Wire Types

Wires are the variables of the Circuit-IR, and fields are their type. Each wire carries an element in some field from the output of one gate to the inputs of one or more later gates.

In the Translation-IR, both fields and wires are variables. Fields are special, because they're both variables and types. Translation-IR wires differ from those of the Circuit-IR, because they don't carry field elements. Instead, they carry identifiers for wires generated in the output Circuit-IR relation.

In the Translation-IR, fields are variables because they must be function parameters in the standard library. It wouldn't be a *standard* library if it were specific to a field. However, this requirement collides with the Circuit-IR's requirement that fields be declared as front matter to the relation to allow TA2 to prepare for each field. To resolve this issue, we add a constraint to the Translation-IR: fields may be declared only in the global scope (easy to scan), but can be passed as constant parameters to function calls.

3.3. Arithmetic and Boolean Gates

In the IR 1.0 revision, a gate set was specified to choose between arithmetic gates (`@add`, `@mul`, etc.) and Boolean gates (`@xor`, `@and`, etc.). This revision does away with the Boolean gates and uses equivalent functionality in *GF(2)*.

Boolean Gate (old)	Arithmetic Replacement (new)
`@xor`	`@add`
`@and`	`@mul`
`@not`	`@addc(x, <1>)`

3.4. Field Conversions

A key goal for Phase II's IR is that multiple fields be allowed in the same relation, and that TA2 be able to perform conversions from one field to another ("Field Switching"). In field switching we consider each wire to be a digit in the base of its prime. When converting from a large field to a small field a single digit must be decomposed into multiple digits of a smaller base, or vice-versa.

The Circuit-IR requires that a specification for conversions be given in the front matter to allow TA2 to prepare for necessary conversions. The conversion specification declares the input and output fields and wire counts. Similarly to field specifications, there is the possibility that the standard library uses conversions that differ by function invocation. To square the standard library with the Circuit-IR's front matter, the Translation-IR is again constrained with the requirement that allowable conversions are specified in the global scope.

Another intricacy of field switching is the particular algorithm for converting numeric values from one field to another. Pseudo-code for a canonical conversion is taken from Wizkit's field-switching collab with PROVENANCE last year. Deviations from this conversion algorithm may be implemented in the Translation-IR as a function wrapping the canonical conversion but may be overridden by backends supporting the deviation naturally.

3.5. Instance and Witness Streams

Each field in a relation will require an instance and witness stream. Each stream is specific to a single field, because we expect that certain extensions and transformations will modify or add to the stream. It should be easier to track one stream per field during translation than to possibly back-fill empty spaces in a stream.

NOTE this proposal currently requires additional thought and clarity on how multiple input streams will be tracked from TA1 through translation and into TA2.

The field starts with single field declaration; then each line has a stream value of the form `< value >;`.

4. The Circuit-IR

The Circuit-IR, in general, is a flat list of gates. It does allow function gates to wrap a sub-list of gates for reuse. Due to the large expected size of relations, and for ease of referencing from the Translation-IR, a wire-numbering scheme is used for the Circuit-IR.

4.1. Field and Conversion Specifications

The Circuit-IR requires that fields and conversions be specified in front matter, between the IR header and the `@begin` keyword.

A field specification indicates the field's characteristic (prime). It also implicitly specifies a field-index, assigned incrementally as each field is specified.

```
// index 0: Boolean
@field 2;
// index 1: 2^61 - 1
@field 2305843009213693951;
// index 2: 2^255 - 19
@field 57896044618658097711785492504343953926634992332820282019728792003956564819949;
```

The conversion specifications may be interleaved with the field specifications. Each conversion specification has the form `@convert(@out: out_field_idx: out_length, @in: in_field_idx: in_length);` Here are a few examples.

```
// Convert Booleans to Mersenne61 and back
@convert(@out: 1:1, @in: 0:61);
@convert(@out: 0:61, @in: 1:1);
// Convert Mersenne61 to 25519 and back
@convert(@out: 2:1, @in: 1:5);
@convert(@out: 1:5, @in: 2:1);
```

4.2. Memory Management

Unlike prior iterations of the Circuit-IR, this revision places strict restrictions on memory management, specifically when consecutive wires may be considered to be stored in contiguous space. Two wires may have consecutive wire-numbers, but live in non-contiguous space.

Each field is given its own numbering space, with wire numbers in the range of $0 \ldots 2^{64}-1$. Most directives will use a field-index parameter to select in which field, and in which numbering space, they will act. For example, `0: $123` and `1: $123` may both be defined, with each wire residing in a different numbering space due to their different fields.

To create wires in contiguous space, the `@new(field_idx: $first ... $last);` directive may be used. It creates space for wires, but does not assign values to them.

```
@new(1: $100 ... $200);
```

The `@delete` directive remains with the form `@delete(field_idx: $first ... $last);`. When deleting contiguous space (allocated with `@new`), the `$first` and `$last` parameters must match the preceding `@new` directive; however, non-contiguous wires may also be deleted.

```
// fail - does not match prior new
@delete(1: $100 ... $110);
// delete the prior new
@delete(1: $100 ... $200);

// assign, but don't @new 1: $53 ... $68

// okay - these wires are not contiguous and may be deleted in any order
@delete(1: $53 ... $60);
@delete(1: $62 ... $68);
@delete(1: $61 ... $61);
```

As with prior IR revisions, each wire in a `@delete` range must be assigned, but not previously deleted. Also, once deleted, a wire may not be reused.

IMPORTANT

Notice that the form of nearly all ranges in the IR is `first … last` rather than `first … length`. Ranges are inclusive on both ends.

4.3. Standard Gates

The form of most standard gates is `$out <- gate_name(field_idx: $left_in, $right_in);`. Other gates have variations on this, and are described as necessary.

- `@add` arithmetic addition
- `@mul` arithmetic multiplication
- `@addc` arithmetic addition by a constant
  - Has the form `$out <- @addc(field_idx: $left_in, < right_constant >);`
- `@mulc` arithmetic multiplication by a constant
  - Has the form `$out <- @mulc(field_idx: $left_in, < right_constant >);`
- Copy the input wire to the output wire
  - Has the form `$out <- field_idx: $left_in;`
- Assign the input constant to the output wire
  - Has the form `$out <- field_idx: < left_constant >;`
- `@instance` and `@short_witness` assign wires using instance or witness stream inputs.
  - Have the form `$out <- @stream_name(field_idx);`

- `@assert_zero`
    - Has the form `@assert_zero(field_idx: $wire);`

4.4. Conversion Gates

Conversion gates enable conversion of wires from one field to another. Conceptually a list of wires in field A is converted to a list of wires in field B. Within the circuit, conversion has the form `out_field_idx: $out_first [... $out_last] <- @convert(in_field_idx: $in_first [... $in_last]);`. The conversion's fields and sizes must match a conversion specification from the front matter.

```
// convert Booleans to a single Mersenne61
1: $0 <- @convert(0: $1 ... $61);
// convert a single 25519 to 5 Mersenne61s
1: $1 ... $5 <- @convert(2: $0);
```

The input list to the `@convert` gate must be either a single wire, or it must be contiguous. The output list may be contiguous, or else the backend should allocate contiguous space for them.

4.5. Function Gates

Function gates define a sub-circuit which may be reused multiple times. The function’s outputs and inputs are given as ranges mapped sequentially, and by field, into the function’s scopes. In the function’s signature, each range is defined by a length and a field index. When the function is invoked, each range is mapped into its scope incrementally from 0.

The remapping process during function invocation is aware of memory contiguity, and should reject ranges which are discontiguous. It should allow non-assigned output ranges to be allocated implicitly.

The function declaration and invocation have the following forms

```
@function(function_name,
    [@out: out_field_idx_0: out_field_count_0 [, out_field_idx_n: out_field_count_n],]
    [@in: in_field_idx_0: in_field_count_0 [, in_field_idx_n: in_field_count_n])`.
  /* gate list */
@end


[out_field_idx_0: $out_first_0 [ ... $out_last_0 ]
  [, out_field_idx_n: $out_first_n [ ... $out_last_n ] ] <- ]
  @call(function_name [, in_field_idx_0: $in_first_0 [ ... $in_last_0 ]
      [, in_field_idx_n: $in_first_n [ ... $in_last_n ] ] ]);
```

Function Gate Example

```
@function(dot_prod_10, @out: 1:1; @in: 1:10, 1:10)
```

```
  // omitted
@end

@new($0 ... $9, 1);
@new($10 ... $22, 1);
// assign $0 ... $19

1:$25 <- @call(dot_prod_10, 1: $0 ... $9, 1: $10 ... $19);
```

Function Declaration Ordering and Recursion

A potential point of contention is the order in which functions must be declared. A natural restriction on the IR is that a function be declared before its invocation. However, there may be debate as to how we define "before invocation". This could be done either in lexical order or in program execution order.

```
@function(a) /* ... */ @end

@function(b)
  @call(a);
@end

@call(b)
```

```
@function(b)
  @call(a);
@end

@function(a) /* ... */ @end

@call(b)
```

The left example is correct in both lexical order and program execution order. The right is only correct in program execution order. There may be a push and pull between complexity in the IR2 translator and in TA2 backends. Lexical order might require the IR2 translator to analyze a function and emit other functions before emitting a size- and field-specific function. However, program execution order might require a TA2 to delay intermediate compilation of function declarations until sub-functions are declared.

4.6. Example

Here is the right-triangle example using the Circuit-IR.

Relation

```
version 2.0.0-wtk;
circuit;
  @field 7 arithmetic;
  @field 127 arithmetic;
  @convert(1:1, 0:1);

@begin
  // mod 7 hypotenuse
  $0 <- @instance(0);
  // mod 7 legs
  $1 <- @short_witness(0);
  $2 <- @short_witness(0);

  // mod 7 is too small to square them
  1:$0 <- @convert(0:$0);
  1:$1 <- @convert(0:$1);
  1:$2 <- @convert(0:$2);

  // square them
  $3 <- @mul(1: $0, $0);
  $4 <- @mul(1: $1, $1);
  $5 <- @mul(1: $2, $2);
  $6 <- @add(1: $4, $5);

  // negate the hypotenuse
  $7 <- @mulc(1: $3, <126>);

  // assert equal
  $8 <- @add(1: $6, $7);
  @assert_zero(1: $8);
@end
```

Instance

```
version 2.0.0-wtk;
instance;
  @field 7 arithmetic;
@begin
  < 5 >;
@end
```

Witness

```
version 2.0.0-wtk;
short_witness;
  @field 7 arithmetic;
@begin
  < 3 >;
  < 4 >;
@end
```

5. The Translation-IR

The Translation-IR is a higher-level IR which is evaluated as a program that produces a circuit. At each (non-public) expression, a gate is emitted in the target — a Circuit-IR relation. Lexical scoping, an expressive type-system, and public conditioned control flow allow for Translation-IR relations to be compact, with reusable elements. A key requirement in making the IR flexible and reusable is the standard library, which specifies common functionality and enables each backend to provide alternative implementations. While the specification calls for the Translation-IR to produce a relation in the Circuit-IR, it should be possible to perform a syntax-to-syntax translation to similarly capable backend-specific IRs, or interpret the Translation-IR directly in ZK.

NOTE

While the Circuit-IR uses the `@` prefix on all keywords, the Translation IR does not. In general, keywords which will emit to the output circuit will be prefixed with `@` (e.g., `@mul`, `@assert_zero`), while keywords with meaning only in the Translation-IR do not have the prefix (e.g., `size`, `for`).

5.1. Libraries and Includes

The Translation-IR should allow for libraries to be included into a relation (or as a sub-component of another library). The included libraries must be listed by their include path, and an identifier is required for moduling members of the library. The syntax for including a library is `include "path" as identifier;`.

```
include "std/vectors.sieve" as vec;
```

When using elements from the library it must be prefixed by the library's identifier. For example, to use a dot product from the vectors library, `vec::dot_prod` rather than just `dot_prod`.

NOTE There are reasons for and against library prefixes, namely avoidance of global namespace pollution versus a small amount of added complexity.

5.2. Type System

The following types are allowed in IR2. In general, values of each type become immutable once they are assigned, upholding the Static Single Assignment principle. The `modifies` keyword may, in some situations, allow values to be replaced.

Type	Description	Specification
`integer`	This holds public intermediate calculations to produce field elements and field attributes.	unbounded, unsigned integer
`field`	These define a field and may be used as the type of *wires*. Fields must be defined in a global scope and passed as arguments to functions.	unbounded and unsigned integer prime, and possibly additional attributes
`size`	Publicly known sizes and indices of other objects.	64-bit unsigned integer
`condition`	A publicly known Boolean condition, used to select if branches. The "Boolean" name is disused to disambiguate public conditions from Boolean circuits.	`true` or `false`
Wire	These reference wires in the output circuit.	a pair of field-index and wire number.

Type	Description	Specification
Tensor	Other types may be wrapped by an array, matrix, or higher-dimensional tensor to aggregate multiple values of the same type. For tensors of a wire type, the tensor holds only a range of wires in the output.	The base type, a list of dimensions, and a range of wire numbers or a list of IR2 values.
`struct`	A `struct` is a compound data type using members with other types. Its behaviors are defined by member functions allowing creation, member access, and mutation.	A list of member variables followed by a list of member functions.

5.2.1. The `field` Type

The reason for holding a distinction between `integer` and `field` is that the Circuit-IR requires all fields to be specified in the front matter. If fully dynamic specification of fields were allowed in the Translation-IR, then whole program analysis would need to be performed on the relation to enumerate fields before generating the circuit. Instead, by restricting `field`s to specification in the global scope, a quick scan will suffice to enumerate all fields in the Circuit-IR's front matter.

The following statement creates a field in the Translation-IR. It may only be used at the global scope of a `translation` or `library` resource. The `[variable-name]` is an identifier and the `[characteristic]` is an integer literal to define the field's prime.

```
@field [variable-name] <- [characteristic];
```

NOTE We use the `@field` keyword in the understanding that so far the SIEVE Program has focused on ZK using prime fields. It remains unclear whether or not alternative field types (or wire types) might be desirable to the program. Both extension fields and rings are of interest to various performers and may be added to the IR.

Field variables (like most other variables) may not be reassigned; however, (unlike most other variables) there are no expressions capable of producing new field values. Instead, a field may be passed as a parameter to a function gate. This way, a relation's fields may be fixed in the global scope and passed through invocations to functions which may vary by field. This allows for very controlled function variations based on fields.

5.2.2. Assignment Status

Due to the used of named return values in functions, variables in the Translation-IR are given an assignment status: either *assigned* or *non-assigned*. Most variables will be assigned immediately. For example, a value is assigned to a local variable. Functions use named output parameters, which will not be assigned at their creation, but which must be assigned before the function completes. In this case, an output parameter knows what type it must accept, but does not carry a value until it is assigned.

Tensors make for a special case of non-assignment when some elements are assigned and other elements are non-assigned. When a tensor is in this state, we call it a partially assigned tensor.

When the last non-assigned element is assigned, the tensor changes from partial assignment to full assignment.

Finally there is a restriction emplaced on when a non-assigned variable may be assigned. It must be assigned in the same scope in which it is created. For example, a function output may not be assigned in a sub-scope of the function; instead, it must be assigned in the function’s outermost scope. However, it could be assigned as the output of a for loop or if statement, emplacing it into a sub-scope.

5.2.3. Tensor Indexing Schemes

Wire tensors, in the Translation-IR, are simply an indexing scheme to produce the appearance of dimensionality over what is actually a flat range. Modifying the indexing scheme to traverse in alternate directions or to select sub-ranges may be useful or even necessary. For example, a for loop may want to assign column-wise to a matrix rather than row-wise.

Unfortunately, this brings up the possibility for an inconsistency between IR2’s and IR0’s memory models. While in IR0 we encounter strict memory contiguity requirements, specifically around function calls, in IR2 we need alterations to a tensor’s indexing scheme — possibly creating gaps or jumps in the range. This is resolved with the `@wire(tensor)` expression which will emit gates to copy the `tensor` into a flat range.

NOTE

Originally, we we considered a few options to resolve this issue:

1. Disallow modified indexing schemes
2. Inline function calls where indexing schemes would cause issues
3. Automatically flatten indexing schemes by copying elements before emitting an IR0 function call
4. Require the frontend to flatten the indexing scheme when necessary

Option 1 was considered undesirable because it lost too much functionality. Option 2 was also considered undesirable because it could have caused blow up in the size of the output, and, more importantly, when using backend plugins it was likely unable to solve the problem. Of the remaining options, option 4 was proposed as the best solution because it gave the frontend better control over potential performance tradeoffs.

5.3. Expressions

Translation-IR expressions can generally be classified as public expressions and gate expressions. Public expressions change the public state of the IR2 program. Gate expressions emit gates to the output circuit and produce references to those gates' outputs in the public state. Naturally, an identifier is also an expression which produces the value of a variable: either a public value or the reference to a private value. There are a few classes of expressions.

- Integer expressions manipulate `integer` and `size` variables.
- Conditional expressions compare `integer`s and `size`s.

- Index expressions select items, sub-ranges, or alternate indexing schemes from tensors.
- Gate expressions emit gates to the output circuit.
- Conversion expressions are a special case of gate expressions which allow for converting wires from one field to another in the output circuit.
- Reserve expressions create non-assigned tensors which may be assigned to later.

5.3.1. Integer Expressions

An integer expression manipulates `integer` and `size` variables. The simplest form of this expression is the numeric literal: `0`, `1`, `2`, etc. Similarly, the identifier of an `integer` or `size` variable is also an integer expression.

More interesting integer expressions have the form `(lhs op rhs)` where `lhs` and `rhs` are sub-expressions and `op` is one of the following:

- `*` multiplication
- `/` integer division
- `%` modulo operation or remainder
- `+` addition
- `-` subtraction

Finally there are two special expressions for converting public types. The `size(sub_expr)` expression will convert an `integer` or `field` value to a `size` mod 2^{64}. The `integer(sub_expr)` expression will convert a `size` or a `field` value to an `integer`. Notice that `field` may be converted to `size` or `integer`, but the reverse is not possible, and the arithmetic expressions do not operate over `field` values.

5.3.2. Conditional Expressions

Conditional expressions are public Booleans and are mainly used for selecting branches of if statements. There are two forms of conditional expressions, in addition to the literals `true` and `false`.

The comparison form will create the condition after examining two sub-expressions with result type `integer` or `size`. They have the form `(lhs op rhs)`, where `op` is one of the following.

- `==` equal
- `!=` not equal
- `>` greater than
- `<` less than
- `>=` greater than or equal
- `<=` less than or equal

The Boolean form will create the condition after examining two sub-expressions both with result type `condition`. They also have the form `(lhs op rhs)`, and the `op` is one of the following:

- `&&` Boolean and
- `||` Boolean or
- `(! sub_expr)` Boolean not

5.3.3. Index Expressions

Index expressions work with tensors and use the square-brackets notation. The simplest index expression would select an element from a vector, `vec[i]`. If instead it indexed a matrix, then it would select a row, `mat[i]`, producing the result as a vector. To select an element in the matrix use a multi-index expression, `mat[i, j]`.

The index expression can go further than selecting just rows and elements. Previously mentioned was altering the index scheme. An obvious alteration would be to select a column. This requires the dimension reordering operator, `^`, which reorders the operand's dimension indicated by position onto a produced tensor's dimension indicated by a numeric constant. For example, `mat[^0, i]` takes the operand's rows at a fixed column to produce a column vector. With a 3x4 matrix, you'll get the following column vectors:

```
[ 0,  1,  2  ]
[ 3,  4,  5  ]
[ 6,  7,  8  ]
[ 9,  10, 11 ]

mat[^0, 0]:

  [ 0, 3, 6, 9 ]

mat[^0, 1]:

  [ 1, 4, 7, 10 ]

mat[^0, 2]:

  [ 2, 5, 8, 11 ]
```

Going a step further `mat[^1, ^0]` would transpose the matrix. Similarly, a sub-matrix could be formed with a ranged index, `mat[0, 0 … 1]` makes a vector with 2 elements from the first row. Or `mat[0 … 1, 0]` takes two elements from the first column. Putting both together, `mat[0 … 1, 0 … 1]` makes a sub-matrix with two elements from the first row and 2 from the second row. Lastly you could transpose and sub-size at the same time, `mat[^1: 0 … 1, ^0: 0 … 1]`. Using the previous example matrix, you'd get the following new matrix.

```
[ 0 3 ]
[ 1 4 ]
```

Most times when transposing or sub-sizing a tensor you get a tensor with gaps or discontiguities in it. Gaps and discontiguities are incompatible with IR0's memory model for function invocations. To pass a tensor containing discontiguities to another function, it must be copied into a contiguous tensor. Because some approaches to proving statements in the Translation-IR may not have contiguity requirements — even translation to the Circuit IR might inline function calls — the `@wire` expression may be used with a discontiguous tensor to emit copies into a contiguous range of wires. In approaches which do not require wire contiguity in their outputs, the `@wire(tensor_with_gaps)` directive may be replaced with a no-op.

5.3.4. Gate Expressions

Most Circuit-IR gates will have lookalikes in the Translation-IR. When a gate expression is evaluated, rather than performing ZK, a new gate is emitted to the circuit, and its output wire number is the expression's result. Most gate expressions will have the form `@gate(lhs, rhs)`, where `lhs` and `rhs` are sub-expressions. The Circuit-IR's field index is omitted, as it is inferred from the expressions' types.

For example, `@mul(wire_a, wire_b)` could emit `$2 <- @mul(0: $0, $1);`. `wire_a` and `wire_b` must have the same field type (rather than a public visibility type), and the translator will emit the field's index. The expression's result would be `$2`, so that subsequent expressions can emit the correct wire number. Similarly, `@mulc(wire_a, int_b)` could emit `$3 <- @mulc(0: $0, <2>)`. In this case `wire_a` must be a wire, and `int_b` must be an `integer`. `int_b` will be emitted as a constant modulo the prime of `wire_a`'s field.

A special Translation-IR gate is `@wire`, corresponding to the copy and assign directives of the Circuit-IR. These enable either an explicit copy or the assignment of a constant via the following two syntaxes.

- `@wire(wire_expr)` duplicates the wire expression to another wire.
- `@wire(field_name: int_expr)` translates the integer expression's result to a constant assignment in the circuit.

Two additional gate expressions are provided to mirror `@short_witness` and `@instance` from the Circuit-IR. They have the forms `@short_witness(field_name)` and `@instance(field_name)`, and they emit themselves to the output circuit.

5.3.5. Conversion Expressions

The conversion expression also mirrors the Circuit-IR's conversion directive. The expression accepts either a vector of wires or a scalar, and produces either a vector of wires or a scalar in a different field. The input and output fields and sizes must match a conversion specification.

Like the field specification, a conversions specification goes in the top-level scope to indicate fields and vector sizes accepted by conversions.

```
@convert(@out: field_name[ vector_len ], @in: field_name[ vector_len ]);
```

`field_name`s are identifiers referring to field specifications and `vector_len` are integer constants indicating the number of elements in a vector. In cases when `vector_len` is 1, a single wire may be used in place of a vector input or output.

The @convert expression may be used anywhere an expression is allowed. It first accepts a specification to indicate its output type. Then it accepts an input parameter.

```
@convert(out_field[ len_expr ], input_expr);
@convert(out_field, input_expr);
```

When len_expr is omitted a single wire is output rather than a vector. The input type and vector len is determined from the input_expr.

5.3.6. Reserve Expressions

Reserve expressions create non-assigned tensors. They cannot be used as arguments to most other expressions, but they can be used with assignment directives.

```
@reserve(type [size, ...])
```

5.4. Assignments

An assignment directive will add a variable to the program's state, and give it a value based on the assignment's expression.

```
new_identifier <- expression ;
```

The result of the expression (and the result's type) are remembered in the new_identifier. In the case that new_identifier corresponds to a non-assigned identifier (such as a yet to be assigned output), its expected type must match the actual type of expression.

Assignment directives may also be used to partially assign a tensor. In this form the left-hand side takes an index expression rather than an identifier.

```
tensor[index] <- expression;
```

5.5. Assert Zeros

In the Translation-IR the @assert_zero directive accepts an expression with a wire result. It will emit an @assert_zero directive to the circuit requiring that the expression's result is zero.

```
@assert_zero(expression);
```

5.6. Functions

Functions allow a block of code to be bundled up and reused later. They are split between a declaration and an invocation. The declaration directive, allowed only in the global scope, creates a function which may be used in a later in an invocation directive.

The declaration consists of a signature followed by a body. The signature is a list of output parameters followed by a list of input parameters. Since wire and tensor types are defined by expressions of field and size types, prior parameters may be used in expressions defining types. Here is the example of a vector multiplier.

```
@function vector_mul(@out: wire F[S] os, @in: field F, size S, wire F[S] ls, wire F[S]
rs)
  /* Omitted */
@end
```

In this example, `field` and `size` are type keywords to describe the input parameters `F` and `S`. Subsequently the input parameters `ls` and `rs` are defined as wires with type `F`, the `[S]` indicating they are vectors. Similarly, a dot product and a matrix product would be defined like this.

```
@function vector_dot(@out: wire F o, @in: field F, size S, wire F[S] ls, wire F[S] rs)
  /* Omitted */
@end

@function matrix_mul(@out: wire F[A, C] os, @in: field F, size A, size B, size C, wire
F[A, B] ls, wire F[B, C] rs)
  /* Omitted */
@end
```

Within the body, Translation-IR directives may be used to transform the inputs into outputs. The function body must assign all output parameters.

To invoke a function a special `@call` directive is used. Invocation is not an expression because functions are allowed to have multiple output parameters. Input arguments are bound to expression results in function order. For each input, the expression's result type must match the associated parameter's declared type. Each output is assigned to an identifier in list order. For example, the invocation of a division and remainder function may appear like this.

```
// assume a and b are wires in field F
q, r <- @call(div_rem, F, a, b);
```

For partial-tensor assignments, index expressions may appear in the output list.

5.7. For Loops

A for loop will repeat a block of code, and is the best way to assign elements to a tensor. On each iteration an element of the output tensor is assigned by assigning to a non-assigned variable created within the body. Variables may also be created within the loop body to iterate across fully assigned tensors. Here is an example loop which would perform vector multiplication.

```
z in wire F[s] zs <- for i (x in xs, y in ys)
  z <- @mul(x, y);
end
```

In this example the `z in wire F[s] zs` clause creates an outer scope variable `zs` with type `wire F[s]`. It also creates a loop-scope non-assigned variable `z` with type `F` which must be assigned within the body. The `x in xs` and `y in ys` clauses will create inner scope variables (`x` and `y`) which traverse the tensors `xs` and `ys`. Naturally this co-traversal creates the requirement that `xs` and `ys` both have the same length (`s`) as the `zs` output. The `i` iterator is a `size` variable created in the loop's scope, starting at 0 on the first iteration and incrementing on each subsequent iteration.

More formally, loop traversal blocks will iterate over the outermost dimension of a tensor. Output traversals may either either traverse newly created tensors with the syntax `inner_variable in type outer_variable` or existing non-assigned tensor variables with the syntax `inner_variable in outer_expression`. In the latter form, which is also used for input traversals, indexing expressions may be used to perform traversals along alternative indexing schemes.

The loop also allows for variables to be modified once on each iteration. This enables the loop to carry a bit of information to the next iteration, which is necessary for sequential loops, such as an iterative summation. This example would perform a dot product.

```
sum <- @wire(F: 0);
for i (x in xs, y in ys) modifies sum
  sum <- @add(sum, @mul(x, y));
end
```

In the case that repetition is needed, but not for traversal of a list, a `for i repeat n` syntax is offered. In this case `n` is a size expression indicating how many iterations should occur.

5.8. If Statements

An if statement assigns a set of variables based on a publicly known `condition` variable. Because the if statement assigns outputs in its outer scope, the `else` branch must always be present, although 0 or more `elif` branches may be present. Here is an example if statement.

```
// assume that a and b are wires in field F and i and j are integers
wire F result <- if (i < j)
  result <- @mul(a, a);
elif (i == j)
  result <- @mul(a, b);
else // (i > j)
  result <- @mul(b, b);
end
```

The `wire F result` clause creates a non-assigned variable within each branch's inner scope. The first `if` or `elif` block whose condition is `true` is the active branch, and its body is executed. When the end of the active branch is reached, the `result` is emplaced into the outer scope.

Instead of creating a new variable in the outer scope, an outer scope non-assigned variable or partially assigned tensor index expression may be used in the output list of the if statement. The if statement may also specify zero or multiple outputs for itself.

5.9. Structs

The Translation-IR allows for `struct`s to represent data structures. Structs have data members but use functions to expose behaviors rather than accessing members directly. This allows the implementation of a struct to vary while its interface remains the same, in case a backend desires to override it with a bespoke implementation.

The struct's syntax uses the `struct` keyword and its name to begin the struct declaration. Next, data members may be listed using `type member_name;` notation, followed by a list of function declarations for their behavior. Behavior functions' input parameters of the struct's own type may be annotated as `modify`, indicating that the behavior will modify its input parameter. The `modify` keyword can be used where a linear type system must be enforced or where a bespoke ZK optimization would modify data in an otherwise constant behavior.

Within struct member functions, special assignment directives are used to access struct members, create new structs, and, where allowed, modify input structs. The special assignment directives are only usable within their own struct.

- `name_in_scope, … <- access(struct_object: name_in_struct, …);`: retrieve member variables from the struct.
- `object_to_modify <- modify(name_in_struct : name_in_scope, …);`: modify members of a struct.
- `object_in_scope <- create(name_in_struct : name_in_scope, …);`: create a new struct.

Use of the struct is allowed only through its behavior functions.

Here is an example.

```
struct Uint32
  wire Bool[32] bits;

  // create a new struct value as return by adding two existing structs
  @function Uint32_add(@out: struct Uint32 o, @in: struct Uint32 l, struct Uint32 r)
    l_bits <- access(l: bits);
    r_bits <- access(r: bits);

    // assign o_bits using a 32-bit adder circuit

    o <- create(bits: o_bits);
  @end

  // edit a current struct value by adding another's value to itself
  @function Uint32_add_to(@in: modify struct Uint32 l, struct Uint32 r)
    l_bits <- access(l: bits);
    r_bits <- access(r: bits);

    // assign o_bits using a 32-bit adder circuit

    l <- modify(bits: o_bits);
  @end

  // create a struct by assignment of 32 booleans
  @function Uint32_bool_create(@out: struct Uint32 o, @in: wire Bool[32] bits)
    o <- create(bits: bits);
  @end
end

// assume a_bits and b_bits have type wire Bool[32]
a <- @call(Uint32_bool_create, a_bits);
b <- @call(Uint32_bool_create, b_bits);
// create c by summing a and b
c <- @call(Uint32_add, a, b);
// add a to b, modifying b
@call(Uint32_add_to, b, a);
```

NOTE The use of rings to demonstrate of `struct` functionality is not intended to advocate for or against the inclusion of rings as a standard library or as a builtin core functionality.

6. Plugins and Other Functionality

Plugins and other functionalities are not a part of the IR's core functionality, but remain as allowable syntax and are generally handled as unspecified behavior within each backend. Plugins (and variances in fields supported) necessitate a configuration negotiation ("JR") where a ZK backend declares to the frontend which fields and plugins it allows. In this section we describe plugins for both the Circuit-IR and the Translation-IR; then we describe hints to guide the Translation-IR compiler and a configuration negotiation for the Circuit-IR.

6.1. Circuit-IR Plugins

Circuit-IR Plugins allow a circuit to refer to specific functionality which is not defined within itself. Instead, the functionality is called by a name which the backend recognizes.

Plugins look like functions; however, the function body has been replaced with a special `@plugin` directive which enables the backend to recognize the plugin. To declare a plugin function, start with the signature of a function. Then use the `@plugin` directive with a comma-separated sequence of identifiers and numeric literals. Invocation will remain the same as for functions.

```
// declare the function signature with a plugin body
@function(vec_mul_4, @out: 0:4, @in: 0:4, 0:4) @plugin(vector, mul, 4);
```

```
// call the vec_mul_4 plugin
0: $8 ... $11 <- @call(vec_mul_4, 0: $0 ... $3, 0: $4 ... $7);
```

Option: Plugin Specification

If this option is taken, the plugin's name must be specified in the specification section before the `@begin` keyword. The plugin's name must be supplied to each `@plugin` directive as the first parameter.

```
@plugin(vector);
@field 127 arithmetic;
@begin
  // Okay! The vector plugin was declared
  @function(/* ... */) @plugin(vector, /* ... */);

  // Not Okay! The matrix plugin was not declared
  @function(/* ... */) @plugin(matrix, /* ... */);
```

Option: Plugin Fields (Types)

If this option is taken, special fields (types) may be declared by the plugin. Wires of a plugin field must be manipulated via plugin functions.

```
// 0: A regular field
@field 127 arithmetic;
@plugin(ring);
// 1: A plugin field using the ring plugin
// presumably this does arithmetic mod 2**7
@field @plugin(ring, base, 2, exponent, 7);
@begin
  // interaction with plugin fields is allowed via plugin functions
  @function(int7_mul, @out: 1:1, @in: 1:1, 1:1) @plugin(ring, mul);
  @function(int7_add, @out: 1:1, @in: 1:1, 1:1) @plugin(ring, add);
```

NOTE If this option is taken, it may be beneficial for the `@field` keyword to be renamed to `@type` or something less controversial.

6.2. Translation-IR Plugins

Plugins in the Translation-IR are designed to allow the Translation-IR to emit plugins in place of a function's body. Much like in the Circuit-IR, these plugins replace the body of a function with an `@plugin` directive containing a comma-separated list. Again the function's signature is given as normal, and its body is replaced with the `@plugin` directive. Within the `@plugin`'s body, both unrecognized identifiers and integer and size expressions are allowed. Unrecognized identifiers will be emitted directly, while expressions will be evaluated and their numeric result emitted.

The following example will emit a plugin function with similar functionality to the Circuit-IR `vec_mul_4` example.

```
// declare the function signature with a plugin body
@function(vec_mul, @out: wire F[s] os, @in: field F, size s, wire F[s] ls wire F[s]
rs)
  // "vector" and "mul" are emitted directly, while "s" emits a size constant
  @plugin(vector, mul, s);

// calling a plugin function reuses the call directive
// assume that a and b have type wire Mersenne61[sz]
c <- @call(vec_mul_4, Mersenne61, sz, a, b);
```

Option: Plugin Specification

This option mirrors the Circuit IR option, and, if taken, the plugin's name must be specified in a top-level scope, although lexical order may be ignored. The plugin's name must be supplied to each `@plugin` directive as the first parameter.

```
// Okay! The vector plugin is declared later in the file
@function(/* ... */) @plugin(vector, /* ... */);

// Not Okay! The matrix plugin is never declared
@function(/* ... */) @plugin(matrix, /* ... */);

@plugin(vector);
```

Option: Plugin Fields (Types)

This option also mirrors the Circuit-IR option, and, if taken, special fields (types) may be declared by the plugin. Plugin functions must be declared for basic interactions with the plugin type. An expected use case for plugin fields is as an alternate implementation of a `struct`. The example given mirrors the `Uint32` example Translation-IR Structs.

```
@plugin(ring);
@field uint32_impl @plugin(ring, base, 2, exponent, 32);

struct Uint32
  uint32_impl val;

  @function(Uint32_add, @out: struct Uint32 o, @in: struct Uint32 l, struct Uint32 r)
    @plugin(ring, add, 32);
```

6.3. IR2 Compilation Hints

Compilation hints would allow libraries and frontends to make hints about optimal compilation of particular functions to the Translation-IR compiler. For example, a library could make the hint that an internal function be inlined into API functions, or that a very large function never be inlined. Or a frontend might suggest to the Translation IR compiler that a for loop is easy to vectorize and to emit calls to vector plugin functions rather than unroll the loop. Or a library might supply a plugin implementation with a particular prime or prime family and a fallback implementation for other primes. Regardless, the `hint(⋯)` syntax should annotate other directives with special instructions. In most cases, it should be easy enough to ignore the hint if it isn't recognized.

6.4. Configuration Negotiation ("JR")

The JR identifies Circuit-IR features and plugins which may be supported and combined by a particular backend. The following elements probably should or could be present in the JR:

- Primes and prime families supported by the backend, along with an order of preference
- List of plugins supported by the backend
- Other requirements and preferences:
    - Whether to emit functions or flatten the entire relation
    - Whether the backend prefers circuit breadth or depth

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001120C0087. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.