P4 HLIR 0.9.30 Specification

Table of Contents

Overview

This document describes the structure of the high-level intermediate representation (HLIR) that the P4 compiler generates based on P4 program input. This HLIR is what is used by the compiler backends to implement a forwarding element that is functionally equivalent to what the

P4 program specifies, and to generate code and APIs for interacting with that forwarding element.

Brief Requirements Review

The frontend of the compiler must produce a set of Python objects as described below. This set of objects:

- Must entirely specify the functionality of a P4 program, including any standard library objects that may be referenced (such as common header types like IPv4, and target-specific primitive actions like `modify_field(...)`)
- Is assumed to be entirely valid and conform to semantic rules established by the official P4 specification
- Should be usable across all backends of the compiler, ranging from the actual hardware target to the C API auto-generator
    - If an abstraction extracted from the P4 program is used across many targets (like the table graph extracted from control functions) it should be in the HLIR and generated by the frontend

Generating the HLIR and accessing its objects

To build the HLIR, use the following Python code:

```
from p4_hlir.main import HLIR
h = HLIR(<list of p4 sources>)
h.build()
```

For each type of P4 object (e.g. `table`), the HLIR defines a Python class (e.g. `p4_table`). For each P4 object (e.g. `table ipv4_lpm`), the appropriate class is instantiated and the instance is placed in a Python `OrderedDict`, using the name of the P4 object as the key and the instance as the value. The dictionaries exist as attributes of the HLIR object.

In the above example, one can access the tables defined in the P4 program using the following code:

```
for table_name, table in h.p4_tables.items():
    pass
```

The following `OrderedDict`'s are exposed by the HLIR. A dictionary whose name is `p4_<name>s` contains Python objects of type `p4_<name>`.

`OrderedDict`'s name	P4 object type
p4_actions	`action`

p4_tables	`table`
p4_conditional_nodes	None, this is used to represent conditions contained in the control flow
p4_action_nodes	None, this is used to represent actions contained in the control flow
p4_action_profiles	`action_profile`
p4_action_selectors	`action_selector`
p4_nodes	None, this is an “abstract” class that is inherited by both p4_table and p4_conditional_node
p4_headers	`header_type`
p4_header_instances	`header` and `metadata`
p4_fields	None, this is used to refer to a field in a `header_instance`
p4_field_lists	`field_list`
p4_field_list_calculations	`field_list_calculation`
p4_parse_states	`parser`
p4_parse_value_sets	`parser_value_set`
p4_parser_exceptions	`parser_exception`
p4_counters	`counter`
p4_meters	`meter`
p4_registers	`register`
p4_blackbox_types	`blackbox_type`
p4_blackbox_instances	`blackbox_instance`

In addition to the above `OrderedDict`’s, the HLIR class also exposes these two attributes:

- `p4_ingress_ptr`: a Python dictionary which maps `p4_node` objects to the corresponding Python set of `p4_parse_state` objects. If a parse state is included in the set, it means that this parse state can return the `p4_node` key as a control flow entry point
- `p4_egress_ptr`: the one `p4_node` control flow entry point for the egress pipeline

The rest of this document describes the different Python classes instantiated from the P4 program, and their attributes.

p4_header

There is one `p4_header` object for each `header_type`/`layout` object in the P4 program. The original P4 name is used as a key in the HLIR object dictionary. Each contains the following attributes:

- **name**
  the header type's name, as taken from the P4 program
- **layout**
  An `OrderedDict` (from the Python `collections` module) representing the order and width of the header type's fields. The key represents the field name, and the value represents the field's width in *bits*. The value can also optionally be the special value `p4_AUTO_WIDTH`, meaning the width was '`*`' in the P4 program. The order of the name:width pairs represents the order in which they appear on the wire.
- **attributes**
  An `OrderedDict` (from the Python `collections` module) representing the order and attributes of the header type's fields. The key represents the field name, and the value is a set of attributes which can include `p4_SIGNED` and `p4_SATURATING`. The order of this dictionary is the same as for *layout*.
- **length**
  The header's length, in *bytes*, as either an `int` or a `p4_expression`. The latter is an object representing an expression referencing the values of the header's fields at runtime.
- **max_length**
  The header's maximum allowable length, in *bytes*, as an `int`

p4_expression

Expressions from the P4 program (header type `length` attributes and conditions in control function `if` statements) are represented as trees of `p4_expression` objects, each of which represents a binary or unary operation with the following attributes:

- **left**
  The left-hand operand of a binary operation. For unary operations, this has the value `None`. May be an `int`, `p4_header_instance`, or `p4_field`.
- **right**
  The right-hand operand of a binary operation, or the sole operand of a unary operation. May be an `int`, `p4_header_instance`, or `p4_field`.

- **op**
  The operation being performed, as one of the following strings:
    - “+”
    - “-”
    - “*”
    - “/”
    - “%”
    - “**”
    - “<<”
    - “>>”
    - “|”
    - “^”
    - “&”
    - “~”
    - “==”
    - “!=”
    - “<”
    - “<=”
    - “>”
    - “>=”
    - “not”
    - “or”
    - “and”
    - “valid"

p4_sized_integer

Inherits from the Python base `int` type. It annotates an `int` with an explicit `width` attribute. If the P4 program explicitly specifies a width, then this value is used and the integer is truncated if needed. If no width is specified in the P4 program, then the `width` attribute is inferred from the integer value. Note that as of now, this is only used for integer field list entries.

p4_header_instance

P4 instances declared with the `header` and `metadata` keywords are both represented by `p4_header_instance` objects. Instance arrays appear as a series of separate `p4_header_instance` objects, along with special `p4_header_instance` objects for special P4 index values like `next` and `latest`. The instance's key in the `p4_header_instances` HLIR dictionary is the string that would be used to refer to it in the original P4 program. For example, the P4 declaration "`header my_type my_inst;`" adds the following pair to the `p4_header_instances` HLIR dictionary:

```
{ "my_inst" : <p4_header_instance object> }
```

While the declaration "`header my_type my_array[3];`" would add the following:

```
{ "my_array[0]" : <p4_header_instance object>,
  "my_array[1]" : <p4_header_instance object>,
  "my_array[2]" : <p4_header_instance object>,
  "my_array[next]" : <p4_header_instance object>,
  "my_array[latest]" : <p4_header_instance object> }
```

Each `p4_header_instance` has the following attributes:

- **name**
  The header instance's name, in the same form that the header would be referenced from P4. From the example above, this would be something like "`my_inst`" or "`my_array[2]`", etc.
- **base_name**
  The header instance's name without any array subscripts. For example, a header instance "`my_array[2]`" would have base_name "`my_array`". This is useful for accessing other headers in the same array, by concatenating the appropriate square-bracketed subscript. For non-array instances, name and base_name are equal.
- **fields**
  An ordered list of `p4_field` objects referring to each field in the instance
- **header_type**
  A reference to the `p4_header` object representing the P4 header type this object is an instance of
- **metadata**
  A boolean set to `True` if the instance was declared with the P4 `metadata` keyword, and `False` if with the `header` keyword
- **index**
  If the instance is part of an array, this attribute contains the instance's array position. Either an `int` or the special values `P4_NEXT` or `P4_LATEST`. If the instance is not part of an array, this attribute has the value `None`.
- **max_index**
  If the instance is part of an array, this attribute contains the size of the array as the maximum allowable integer index into it. For example, an array declared as "`header my_tag my_array[5]`" would have a max_index of 4. If the instance is not part of an array, this attribute has the value `None`.
- **virtual**
  A boolean value. If `True`, the instance is a special reference (like `P4_NEXT` or `P4_LATEST`) and thus does not need space allocated in the PHV. Otherwise, the

instance is allocated space in the PHV. Elements of instance arrays with integer indices are not virtual, and neither are instances that are not part of arrays.

- **initializer**
  A dictionary mapping field names (as present in the associated header type's layout) to integer values. The values are written to the corresponding fields between packets. Only metadata header instances will have initializers.

p4_field

There is one `p4_field` object for every field of every header instance in the P4 program. The field's key in the object collection is the string that would be used to refer to the field in P4 (eg, an IP ttl would be retrieved with "`my_ipv4_instance.ttl`" while an MPLS label in a stack would be retrieved with "`my_mpls_stack[3].label`"). Each has the following attributes:

- **name**
  The field's name (eg, "`my_mpls_stack[3].label`" would have the name "`label`")
- **instance**
  A reference to the `p4_header_instance` object that this field belongs to
- **width**
  An `int` recording the field's width, in bits. If the field is of variable-width, has the special value `P4_AUTO_WIDTH`.
- **attributes**
  A `set` recording the field's attributes. It is a subset of {`P4_SIGNED`, `P4_SATURATING`}
- **offset**
  An `int` recording the field's starting offset in the parent header instance, in bits. If there is a variable-width field in the header and normal fields come after it, those normal fields at the end have *negative* offsets indicating the distance in bits between the start of that field and the end of the header.
- **default**
  If the field is part of a metadata instance, an `int` recording the initial value the field should have before being written to. If the P4 program did not specify an initial value in the metadata instance declaration, the default `default` is `0`. If the field in question is part of a non-metadata header instance, the value is `None`.
- **ingress_read, ingress_write, egress_read, egress_write**
  Boolean flags indicating accesses to this field during ingress and egress pipelines.
- **calculation**
  If the field has been bound to field list calculations, this attribute is an ordered list of tuples with values, in order:
  a. a Python string which can either be "update" or "verify", indicating whether the field value should be verified against the calculation, or updated from the calculation
  b. the `p4_field_list_calculation` producing the value

c. the boolean condition expression deciding whether to perform the update or verify operation, as a `p4_expression` object (or None if no condition)
If the field has not been bound like this, the attribute's value is an empty list.

p4_field_list

There is one `p4_field_list` object for each `field_list` defined in the P4 program. The original P4 name is used as a key in the corresponding HLIR dictionary. Each contains the following attributes:

- **name**
  The field list's name.
- **fields**
  An ordered list of references to the `p4_field` objects contained by the field list. May also contain `p4_sized_integer` objects and the special value `P4_PAYLOAD`. Elements in this list may not be unique.

  If the P4 object contained whole headers or other field lists, they have been expanded out here into all of their component fields (recursively expanded, in the case of field lists referencing other field lists). Recursive/cyclic references between field lists are detected and disallowed by the frontend.

p4_field_list_calculation

There is one `p4_field_list_calculation` object for each `field_list_calculation` defined in the P4 program. The original P4 name is used as a key in the corresponding HLIR dictionary. Each contains the following attributes:

- **name**
  The calculation's name.
- **input**
  An ordered list of references to the `p4_field_list` objects included in the calculation
- **output_width**
  An `int` representing the width of the calculation's result in bits
- **algorithm**
  A string representing the calculation algorithm to use. Could be anything, and it's up to the target to check, but the P4 spec defines these common name/algorithm pairs:
  - "**xor16**"
    The XOR of the input bytes, taken two at a time.

    - **"csum16"**
      The IPv4 header checksum described in https://tools.ietf.org/html/rfc791#page-14
    - **"crc16"**
      See http://en.wikipedia.org/wiki/Crc16
    - **"crc32"**
      See http://en.wikipedia.org/wiki/Crc32
    - **"programmable_crc"**
      An arbitrary CRC polynomial. See http://en.wikipedia.org/wiki/Cyclic_redundancy_check.

p4_parse_state

There is one `p4_parse_state` object for each `parser` function defined in the P4 program. The original P4 name is used as a key in the corresponding HLIR dictionary. Each contains the following attributes:

- **name**
  The parse state's name, as taken from the P4 program
- **call_sequence**
  An ordered list of the operations performed in the parse state. These include header instance extractions, metadata field writes, and counter operations. Each list element is a tuple of the form (operation, arg, [further_args, …]) where the operation is a special enum value that dictates the meaning of the following arguments. The different values for operation are:
    - `parse_call.extract`
      A P4 `extract()` operation. The next tuple element is a reference to the `p4_header_instance` being extracted to
    - `parse_call.set`
      A P4 `set_metadata()` operation. The second tuple element is a reference to the `p4_field` being written to. The third tuple element is the value to write, which is either:
        - An integer
        - A (bit_offset, bit_width) tuple representing a call to `current()`
        - A `p4_field` reference from a previously extracted header
    - `parse_call.method`
      A method call on a blackbox instance. The second tuple element is a reference to `p4_blackbox_method` object being invoked, while the third element is a list of argument values being passed to the method.
- **branch_on**
  Either:
    - The empty list [], in which case the branch is unconditional and the branch_to attribute will contain a single entry with the default key

    - A list of fields to be concatenated together into a compound word to branch on, with smaller list indices corresponding to the **more** significant side of the compound word. Each element is either:
        - A `p4_field` reference from a previously extracted header
        - A `(bit_offset, bit_width)` tuple representing a call to `current()`
- **branch_to**
  A `OrderedDict` representing the `case:destination` pairs of the original P4 select function.
  The key may be:
    - An `int`
    - A `(value, mask)` tuple where both elements are `int`s
    - A `p4_parse_value_set` object
    - The special value `P4_DEFAULT`

  The value may be:
    - A `p4_parse_state` object
    - A `p4_table` or `p4_conditional_node` object, in which case the branch is terminal and the parser passes control to the indicated point in the match+action pipeline
    - A `p4_parser_exception` object, in which case the branch is terminal and the packet will either be dropped or passed on to the match+action pipeline
- **prev**
  An OrderedSet listing all parse states that can branch to this one.

p4_parse_value_set

There is one `p4_parse_value_set` object for each `parser_value_set` object defined in the P4 program. The original P4 name is used as a key in the object collection. Each contains the following attributes:

- **name**
  The value set's name, as taken from the P4 program

p4_action

There is one `p4_action` object for each `action` function defined in the P4 program. The original P4 name is used as a key in the corresponding HLIR dictionary. Each contains the following attributes:

- **name**
  The action's name, as taken from the P4 program
- **signature**
  An ordered list of the parameter names the function accepts
- **signature_widths**
  For P4-programmer-defined actions, each element in this list is an `int` recording the bit width of the action's argument at the corresponding index in the `signature` attribute. Use to determine the width of table entry data when the action is invoked directly by a table. Values in this list may be `None` if the argument is never used, or if the action is a primitive.
- **signature_flags**
  For P4-programmer-defined actions, this attribute is an empty dictionary. For primitives, this dictionary maps parameter names (from the `signature` attribute) to a nested dictionary of parameter properties. These properties can include:
    - type
      A Python set of the acceptable argument types for the parameter. In most cases this is some subset of `p4_field`, `int`, and `p4_table_entry_data`. The `int` type indicates an immediate value provided by the programmer within the P4 program itself, while `p4_table_entry_data` indicates a value provided at runtime by the matching table entry which invoked the action. This key must be present with a non-empty set for every parameter in the primitive.
    - access
      An indication about whether the argument is written to or read by the primitive (eg, whether it is a source or destination field). If present, one of the special values `P4_READ`, `P4_WRITE`, or `P4_READ_WRITE`. If not present, it is assumed to be `P4_READ_WRITE`.
    - optional
      A boolean indicating whether or not the argument is optional. If not present, it is assumed to be `False`. Once a parameter has been marked as optional, all parameters following it in the `signature` must also be marked as optional.
    - data_width
      Specifies how to infer the width of the current parameter, should it end up being provided by table data. Can either be an integer, in which case this integer represents the width in bits of the parameter, or a string referring to another parameter whose width can be inferred (e.g. the other parameter can be bound to a header instance's field, whose width is known at compile time). This is how the frontend populates the signature_widths attribute described above.

      For example, in the action `add_to_field (dst, value)`, the `value` parameter would have property `"data_width":"dst"`. If the action was called with a 48-bit field as its `dst` and `value` to be specified at runtime, the API generator could use this property to infer that the API must accept a 48-bit value when installing table entries that invoke the action.

- **call_sequence**
  A list of tuples, each representing a call to another action, ordered as they were written in the original code. Each tuple contains, in this order:
    - The `p4_action` object being called. This may also be a `p4_blackbox_method`, which is a child type of `p4_action`.
    - An ordered list of the arguments being passed to the action. This list is guaranteed to contain at least the target's required parameters, but may not contain values for all optional parameters. Elements of this list may potentially be:
      - Python objects (`int`s, `p4_field`s, etc)
      - References to the calling action's own arguments, as `p4_signature_ref` objects. These have an `idx` attribute which specifies the argument being referenced
- **flat_call_sequence**
  Similar to the `call_sequence` attribute, except all calls to non-primitive actions have been recursively expanded such that the list is entirely composed of primitives and their arguments.

  Each tuple in the list also has a third element reporting the call stack originally taken to get to this primitive, for error reporting purposes. This element is in the form of a list of (`p4_action`, `int`) tuples where the `p4_action` records a parent action and the `int` records the position of the call in that action's unflattened call sequence.

  Actions with empty call sequences are implicitly action primitives.

  Actions are only validated if they are invoked directly or indirectly by a table. If they are not, the action is effectively unused and is trimmed out before the HLIR is presented to the compiler backend.

  It is guaranteed by the frontend that any primitive arguments that are not of type `p4_table_entry_data` have been bound to concrete values at compile-time.

p4_node

There is one `p4_node` object for each table and control flow condition defined in the P4 program. This class has 2 subclasses: `p4_table` and `p4_conditional_node`. This class defines the following attributes:

- **name**
  see description for `p4_table` and `p4_conditional_node`
- **next_**
  see description for `p4_table` and `p4_conditional_node`

- **control_flow_parent**
  A string recording the name of the inner-most control flow function containing this node in the original P4 program, for debugging and error reporting purposes.
- **conditional_barrier**
  A Python tuple with 2 members. The first member is the innermost `p4_node` object on which the execution of the current `p4_node` is dependent. The second member indicates what the output of that `p4_node` needs to be in order for the current `p4_node` to be executed. This second member can either be:
  - a boolean value if the barrier is a `p4_conditional_node`,
  - a Python `set` of `p4_actions` objects,
  - Python strings `"hit"` or `"miss"`
- **dependencies_to**
  A Python dictionary whose keys are the `p4_node` objects on which this node has a dependency. The associated key is a `Dep` object describing the type of dependency.
- **dependencies_for**
  A Python dictionary whose keys are the `p4_node` objects for which this table is a dependency. The associated value is a `Dep` object describing the type of dependency. We can note that if `table_1` is in `table_2.dependencies_to` then `table_2` is in `table_1.dependencies_for`.
- **base_default_next**
  A reference to the `p4_node` which unconditionally comes next in the control flow according to the P4 program. This is essential for P4 tables (not so much for conditions) as the runtime may not configure a default action, and the switch needs to know which node to go to in case of table miss (this information cannot be easily retrieved just by using the table graph, thus the need for this attribute).

p4_table

There is one `p4_table` object for each `table` defined in the P4 program. The original P4 name is used as a key in the object collection. `p4_table` is a subclass of `p4_node`. It defines the following additional attributes:

- **name**
  The table's name, as taken from the P4 program
- **match_fields**
  A list of `(field, type, mask)` tuples where the field is a reference to the `p4_field` being used in the match key, the type is a `p4_match_type` enum value recording the type of match, and the mask is either an `int` or `None` representing a static mask that is ANDed with the field, regardless of match type, before actually matching. The list may be empty. If the match type is `valid`, `field` will actually refer to a `p4_header_instance`. The type value can be one of the following:

    - `p4_match_type.P4_MATCH_EXACT`
    - `p4_match_type.P4_MATCH_TERNARY`
    - `p4_match_type.P4_MATCH_LPM`
    - `p4_match_type.P4_MATCH_RANGE`
    - `p4_match_type.P4_MATCH_VALID`
- **actions**
  A list of `p4_action` references representing the possible actions the table can invoke because of a match. For every packet presented to the table, one of these actions *must* be invoked on it (even on a miss).
- **action_profile**
  A `p4_action_profile` reference for tables with action indirection. `None` if the attribute was not present in the P4 table declaration.
- **min_size**
  An `int` specifying the hard requirement minimum number of entries that must be installable in the table. `None` if the attribute was not present in the P4 table declaration.
- **max_size**
  An `int` specifying the upper bound maximum number of entries the programmer expects to be installed into the table. `None` if the attribute was not present in the P4 table declaration.
- **next_**
  This attribute can be one of two objects:
    - An `OrderedDict` which maps actions that the table can execute to the next node in the table graph which must follow that action (either a `p4_table` or a `p4_conditional_node`, see below). A table is guaranteed to have one key for each element in its `actions` attribute. Values may be `None`, meaning that the packet has reached the end of the current pipeline and should enter the queueing or egress systems.

      The dictionary order is arbitrary and not guaranteed to be consistent across compiler runs.
    - An `OrderedDict` which maps Python strings `"hit"` and `"miss"` to the next nodes in the table graph which must follow a table hit and a table miss. Values may be `None`, meaning that the packet has reached the end of the current pipeline and should enter the queueing or egress systems.

      The dictionary order is arbitrary and not guaranteed to be consistent across compiler runs.
- **support_timeout**
  A boolean value. `True` indicates that the table supports ageing. The runtime is in charge of exposing APIs to set the time-to-live value for entries.
- **attached_counters**
  The `p4_counter` objects bound to this table.

- **attached_meters**
  The `p4_meter` objects bound to this table.
- **attached_registers**
  The `p4_register` objects bound to this table.

p4_action_node

If P4 actions and blackbox methods are called directly from a control function, they are represented in the table graph as `p4_action_node` objects. This class is a subclass of `p4_table` and effectively looks like a table with an empty match key and one action. It defines the following additional attributes:

- **name**
  A unique string name assigned to the conditional node, of the form "_action_N", where N is an arbitrary integer
- **args**
  A list of values bound to each argument in the action call

p4_conditional_node

P4 control functions do not exist in the HLIR. The flow of a P4 program is *entirely* represented by the set of `p4_tables` and special `p4_conditional_node` classes. Each conditional branch (if statement) in the P4 program is modeled by a `p4_conditional_node` object. `p4_conditional_node` is a subclass of `p4_node`. It defines the following additional attributes:

- **name**
  A unique string name assigned to the conditional node, of the form "_condition_N", where N is an arbitrary integer
- **condition**
  The boolean condition expression deciding whether to take the True path or the False path, as a `p4_expression` object
- **next_**
  An `OrderedDict` mapping possible result values of the `condition` expression to the table graph node (a `p4_table` or `p4_conditional_node`) which is visited in that case. Presently, since only boolean conditions are allowed, the only possible keys are `True` and `False`. A node is guaranteed to have both keys, though their values may be `None` (indicating an exit from the current pipeline).

  If either code block of the branch was empty, the appropriate `next` value points to the

node which followed the whole statement in the original control flow program.

The `name` attribute of each node is used as a key in the `p4_conditional_node` object collection.

p4_action_profile

- **name**
  The action profile's name, as taken from the P4 program
- **actions**
  A list of `p4_action` references included in this action profile.
- **size**
  An `int` specifying the upper bound maximum number of entries the programmer expects to be installed into the table.
- **selector**
  The `p4_action_selector` that can be used to select a member (i.e. an action entry) from within a group in the action table. `None` if no selector is to be used with the action profile.

p4_action_selector

- **selection_key**
  The `p4_field_list_calculation` to use to choose members from groups.
- **selection_mode**
  Either `"resilient"` or `"non-resilient"`.

p4_counter

There is one `p4_counter` object for each `counter` object defined in the P4 program. The original P4 name is used as a key in the object collection. Each contains the following attributes:

- **name**
  The counter array's name, as taken from the P4 program
- **type**
  Either the special value `P4_COUNTER_BYTES` or `P4_COUNTER_PACKETS`, indicating whether the counter counts bytes or packets
- **min_width**
  The minimum width in bits, as an `int`, that the compiler must allocate per cell in the counter array

- **saturating**
  A boolean recording whether or not the counters should saturate

The number of cells in the array is determined either through the binding attribute or the instance_count attribute. The frontend guarantees that exactly one will be defined, and the other will have the value `None`:

- **binding**
  A tuple `(binding_type, table_ref)` where `binding_type` is one of the special values `P4_DIRECT` or `P4_STATIC`, and `table_ref` is a reference to the `p4_table` object the counter array is bound to. If direct mapped, the frontend guarantees that this counter object is not referenced by any action in the program. If statically mapped, the frontend guarantees that this counter object is only referenced by actions invoked by the table it is bound to.
- **instance_count**
  The number of cells in the array, as an `int`.

p4_meter

There is one `p4_meter` object for each `meter` object defined in the P4 program. The original P4 name is used as a key in the object collection. Each contains the following attributes:

- **name**
  The meter array's name, as taken from the P4 program
- **type**
  Either the special value `P4_COUNTER_BYTES` or `P4_COUNTER_PACKETS`, indicating whether the meter is measuring bytes or packets

The number of cells in the array is determined either through the binding attribute or the instance_count attribute. The frontend guarantees that exactly one will be defined, and the other will have the value `None`:

- **binding**
  A tuple `(binding_type, table_ref)` where `binding_type` is one of the special values `P4_DIRECT` or `P4_STATIC`, and `table_ref` is a reference to the `p4_table` object the meter array is bound to. If direct mapped, the frontend guarantees that this meter object is not referenced by any action in the program. If statically mapped, the frontend guarantees that this meter object is only referenced by actions invoked by the table it is bound to.
- **instance_count**
  The number of cells in the array, as an `int`.
- **result**
  The field where the meter status (usually called a color) is stored.

p4_register

There is one `p4_register` object for each `register` object defined in the P4 program. The original P4 name is used as a key in the object collection. Each contains the following attributes:

- **name**
  The register array's name, as taken from the P4 program
- **saturating**
  A boolean recording whether or not the registers should saturate
- **signed**
  A boolean recording whether or not the registers are signed

The register layout is determined either through the layout attribute or the width attribute. The frontend guarantees that exactly one will be defined, and the other will have the value `None`:

- **width**
  Each register cell in the array is a bit string with this bitwidth
- **layout**
  This is a reference to a `header_type` object. Each register cell in the array has the same layout as the `header_type`.

The number of cells in the array is determined either through the binding attribute or the instance_count attribute. The frontend guarantees that exactly one will be defined, and the other will have the value `None`:

- **binding**
  A tuple `(binding_type, table_ref)` where `binding_type` is one of the special values `P4_DIRECT` or `P4_STATIC`, and `table_ref` is a reference to the `p4_table` object the register array is bound to. If direct mapped, the frontend guarantees that this register object is not referenced by any action in the program. If statically mapped, the frontend guarantees that this register object is only referenced by actions invoked by the table it is bound to.
- **instance_count**
  The number of cells in the array, as an `int`.

p4_parser_exception

There is one `p4_parser_exception` object for each `parser_exception` object defined in the P4 program. The original P4 name is used as a key in the object collection. Each contains the following attributes:

- **name**
  The `parser_exception` name, as taken from the P4 program
- **set_statements**
  An ordered list of the P4 `set_metadata` operations performed in the parser exception. Each operation is represented by a 3-tuple. The first element is always `parse_call.set`. The second tuple element is a reference to the `p4_field` being written to. The third tuple element is the value to write, which is either:
  - An `int`
  - A `(bit_offset, bit_width)` tuple representing a call to `current()`
  - A `p4_field` reference from a previously extracted header
- **return_or_drop**
  The action to take when this exception is raised. It can either be the special value P4_PARSER_DROP if the packet is dropped or a `p4_node` (`p4_table` or `p4_conditional_node`) object if the packet needs to be submitted to a control flow node.

p4_blackbox_type

There is one `p4_blackbox_type` object for each `blackbox_type` object defined in the P4 program. The original P4 name is used as a key in the object collection. Each contains the following attributes:

- **name**
  The `blackbox_type` name, as taken from the P4 program
- **attributes**
  A dictionary mapping the names of the type's supported attributes to their corresponding `p4_blackbox_attribute` objects
- **methods**
  A dictionary mapping the names of the type's supported methods to corresponding `p4_blackbox_method` objects
- **instances**
  A record of all blackbox instances in the program with the given type. Maps each instance name to its corresponding `p4_blackbox_instance` object

p4_blackbox_instance

There is one `p4_blackbox_instance` object for each `blackbox` object defined in the P4 program. The original P4 name is used as a key in the object collection. Each contains the following attributes:

- **name**
  The `blackbox_instance` name, as taken from the P4 program

- **blackbox_type**
  A reference to the `p4_blackbox_type` for the given instance
- **attributes**
  A dictionary mapping the names of attributes to their values, which depending on the attribute type specified by the `p4_blackbox_type` will be:
  - a reference to another HLIR object
  - an `int`
  - a `str`
  - a `p4_expression`
  - a `p4_sized_integer`

  Optional attributes which were omitted will be absent from this dictionary, while required ones are guaranteed by the frontend to be present.
- **methods**
  A dictionary mapping the names of the instance's methods to their corresponding `p4_blackbox_method` objects. These are copies of the `p4_blackbox_type`'s method objects, and are the ones actually pointed to by other HLIR nodes containing blackbox method calls.

p4_blackbox_method

There is one `p4_blackbox_method` object for every method of every blackbox instance in the P4 program. There are additionally `p4_blackbox_method` objects for each method prototype specified by a blackbox *type*, though these don't appear in any object collections and are primarily used as templates to create each instance's method objects.

This class is actually a subclass of `p4_action`, and appears in the `p4_actions` object collection with names of the format "`instance_name.method_name`". While they otherwise look and behave like action primitives, they also have the following additional attributes:

- **parent**
  A reference to the parent `p4_blackbox_instance` object that owns this method
- **access**
  Blackbox methods are likely to access the attributes of the blackbox instance. The `access` member is a dictionary mapping the type of access (either `"reads"` or `"writes"`) to the set of attributes accessed.

p4_blackbox_attribute

There is one `p4_blackbox_attribute` object for each attribute declared within a `blackbox_type` in the P4 program. There is no object collection for blackbox attributes - they must be accessed through their parent HLIR `p4_blackbox_type`. Each contains the following

attributes:

- **name**
  The attribute's name
- **parent**
  The attribute's parent `p4_blackbox_type`
- **optional**
  A boolean indicating whether or not the attribute is optional
- **value_type**
  An opaque specification of the argument's type.
  TODO: In the near future, expose with more transparency to the HLIR

Dep

The `Dep` Python class defines the following attributes:

- **from**
  The parent node for the dependency.
- **to**
  The child node for the dependency.
- **fields**
  A list of `p4_field` objects which induce the dependency between the two nodes.
- **dependency_type**
  One of `Dep.REVERSE_READ`, `Dep.SUCCESSOR`, `Dep.ACTION`, `Dep.MATCH`.

`Dep` should be used as an abstract class. It defines the following subclasses which can be instantiated:

`ReverseReadDep`

`SuccessorDep`

- `value`: the value that conditions whether the child node will be executed. It is a boolean if `from_` is a `p4_conditional_node`, a `set` of `p4_action` objects (or `"hit"` / `"miss"`) if `from_` is a `p4_table` or `p4_action_node`

`ActionDep`

`MatchDep`

In all cases, these objects have filename and lineno attributes recording where in the original code they showed up. An optional doc attribute contains a programmer-provided docstring.

The frontend guarantees that the HLIR contains a parse state entitled `start`, and that the parse graph anchored at `start` exits to a table graph node at some point. A global variable `p4_ingress_ptr` stores a `dict` whose keys are all possible table graph nodes the parser can exit to. The key's value is a `set` of the parse states which can branch to it. Another global variable, `p4_egress_ptr`, points directly to the first table graph node in the egress pipeline, if it exists (`None` if not).