The P4 Language Specification

Version 1.1.0 Addendum

January 27, 2016

The P4 Language Consortium

This addendum captures some of the technical discussions going on in the P4 Language Design working group, for information about possible future directions. *It does not contain any official specifications of the P4 language.* The working group may or may not take the technical ideas proposed here into a future P4 specification.

1 Architecture-language separation

The current P4 specification defines the language relative to an abstract forwarding model with a specific architecture, as described in Section **??**. The P4 Language Design group is working towards the separation of language features from architecture features, and this addendum gives a summary preview of ideas under consideration. The aim is to ensure that the P4 language specification itself is no longer bound to a particular target architecture. Instead, the intent is to allow target providers to introduce different target architectures (which are heterogeneous compositions of programmable and non-programmable regions) for their targets, while allowing the users of the targets to program any of the programmable regions in the targets in P4. In doing this, an important goal is to make it possible to write portable P4 code, with ways to easily combine that portable code into programs for specific architectures. While target-specific extensions will be allowed, the mission is to encourage portable programs and portable implementations. To ensure portability, the P4 working groups will define one or more standard architecture(s). A P4 program written for a standard architecture will be portable across all the targets conforming to the standard architecture. It is expected that target vendors will implement extensions with the expectation of (some of) those extensions may make their way into future versions of the standard architecture.

1.1 Targets

machine that can run a P4 program is called *target*. While P4 provides a standard language for describing the logic within programmable regions of a forwarding element, the programmable regions that are actually available and the data flow between those regions can vary from target to target. For example, one target may consist of a parser, ingress match+action pipeline and egress match+action pipeline, connected in

sequence. Another target may consist of several parser-pipeline pairs, which the packet may flow through in any order by setting appropriate control signals.

With architecture-language separation, the P4 language itself would address only the contents of each programmable region. Then the overall P4 framework would provide the setting for target providers and standards bodies to define architectures that complement the standard architecture(s). To accomplish this, each target would conform to a *Target Architecture*, specified partially as a collection of P4 code, and partially as a set of specifications that describe the P4-programmable regions of the target and how those regions interact with each other. The latter specifications may involve non-P4 (or possibly extended P4 in the future) rigorous written descriptions and/or simulation models. Examples of programmable regions would be packet parsers, and pipelines of tables and actions. The existing P4 abstract forwarding model would be one example of a standard architecture.

In addition to the Target Architecture specification, there is a *Target-Specific Library* specification. This provides definitions of available P4 extern object types (Section **??**), which represent the processing capabilities of the target beyond the standard P4 primitive actions. The functional specifications of these objects may involve non-P4 (or possibly extended P4 in the future) rigorous written descriptions and/or simulation models. Examples of these capabilities could include arithmetic functions and checksum generators.

1.2 Target Architecture Structure

The P4 portion of a target architecture description provides *prototypes* for the programmable regions of the target. These prototypes specify the input and output interfaces to each region, including the format of metadata passed across each interface. These interfaces form the connection between the region of P4 code in question and the surrounding non-P4-programmable regions. For instance, a region of code may receive intrinsic metadata reporting a packet's ingress port and length, and may write intrinsic metadata controlling the packet's egress port and queue priority.

Together with this P4-described portion, the additional non-P4 specification clarifies the meaning of the context for the P4 portion of the architecture and explains how these portions fit together. Initially, this is envisaged to be mostly human-language documentation and visual diagrams to show the flow of data between programmable regions, though it may also contain pseudocode or simulation models to specify rigorously the behavior of logic not expressible in P4. Looking further out, there has already been experimentation with some extension of P4 itself, to allow rigorous specification of the interaction between regions in terms of established P4 mindset and terminology.

1.3 Target Architecture Selection

A program's target architecture is selected by including that architecture's P4 prototypes in the source code, and then writing structures that conform to the prototypes it specifies. These structures can make use of the extern object types that are provided in the accompanying target-specific library.

No one architecture is mandated by the P4 spec, and a given physical target may support multiple architectures. Some architectures may be written by a target provider and highly specialized to the underlying machinary, while others may be standardized and intentionally abstract to allow greater portability and ease-of-use. A particular example of the latter is that a standard architecture will be defined based on the existing abstract forwarding model.

An important aspect is that all P4 programs written for a given architecture are portable across all targets that faithfully implement that architecture (assuming that enough resources are available). P4 conformance of a target is defined as follows: if a specific target supports a given target architecture, then a program written to that architecture and executed on the target must provide exactly the same behavior as the same program executed on an abstract machine with infinite resources.

In general, P4 programs are not expected to be portable across different architectures. For example, executing a P4 program that controls packet broadcast by writing special intrinsic metadata will not work on a target that provides no such intrinsic metadata. Further, particular targets may not support fully some P4 language constructs (for example, some targets may not support features necessary for IPv4 options processing or arbitrary-length stacked protocol headers). Ideally any restrictions on the P4 language imposed by a specific target should be clearly documented by the target architecture. At the very least, restrictions have to be conveyed to P4 programmers using clear compiler error messages when attempting to compile programs that use unsupported features.

1.4 Programmable blocks

Programmable blocks are user-defined blocks of P4 code that can be instantiated multiple times within a program, and interact with the enclosing target architecture by occupying its programmable regions. Each instance of a programmable block matches a P4-described prototype in the architecture specification.

1.4.1 Programmable block types

A programmable block type is comprised of a signature and code body. The body forms a new scope that can contain any normal P4 declaration. The enclosed code is lexically

scoped and additionally has access to the external metadata parameters declared by its input-output signature.

Similarly to header types for example, the objects declared inside a programmable block type do not actually "exist" inside the program until the block is *instantiated*. In this sense, a programmable block type is declaring a "template" of P4 code that can be stamped down into the program.

1.4.2 Programmable block instances

An instance of a programmable block type represents concrete resource declarations of the contents of the block. Because of this, blocks cannot be instantiated dynamically at run time: they are static, compile-time declarations.

When creating an instance, the programmer must bind all of the input-output parameters in the type's signature either to constants or other object names that are currently in scope.

Multiple instances of the same block type create completely separate instances of the type's component objects which the surrounding architecture and/or a runtime API can refer to using dotted notation.

1.4.3 Programmable block prototypes

Target architectures use programmable blocks to segment P4 code into the various programmable regions of the underlying target. The architecture specifies the prototypes of the blocks it expects to be filled in by the program. These prototypes specify the signature of a block but leave its implementation undefined. They are expected to be paired with a concrete programmable block declaration that has a matching signature.

Prototypes may also include type variables, which are resolved to concrete types when the prototype is paired with its implementation. The identifiers in a prototype's type variable list are available as valid types for the parameters in the prototype's signature. These type variables provide a mechanism for architectures to pass user-defined types of header instances between P4 code blocks without mandating ahead of time what those structs are.

A target architecture may specify several prototypes for identical underlying resources (such as n prototypes for n separately programmable yet functionally identical hardware parsers). A program may use different instances of the same programmable block to satisfy all of the identical prototypes expected by the architecture.

While not explicitly disallowed, P4 programmers are unlikely to find much benefit from writing their own prototypes. Their utility is in target architecture specification only.

1.5 Standard Library

The P4 portion of a target architecture description provides definitions of its extern object types. To promote portability of P4 programs, alongside the standard set of primitive actions, there is a standard library of extern object types for common packet processing operations. While targets may provide target-specific libraries that offer more specific and finely-tuned functionality, this library provides more generalized functionality that all targets should be able to support.

In addition, the definition of a standard library of extern object types assists in simplifying the P4 language, since the function of many constructs currently in the language can be delegated to extern objects, thus simplifying the core P4 language significantly.

1.5.1 Primitive Actions

The primitive actions are standard and expected to be supported by *all* targets, regardless of the target architecture being used. The list of library actions may be a subset of the current P4 list which is given in Section **??**:

Name	**Summary**
`add_header`	Add a header to the packet's Parsed Representation
`copy_header`	Copy one header instance to another.
`remove_header`	Mark a header instance as invalid.
`modify_field`	Set the value of a field in the packet's Parsed Representation.
`no_op`	Placeholder action with no effect.
`push`	Push all header instances in an array down and add a new header at the top.
`pop`	Pop header instances from the top of an array, moving all subsequent array elements up.

Table 1: Standard Primitive Actions

1.5.2 Parser Exceptions

The parser exceptions are standard, regardless of target architecture. The prefix "pe" stands for parser exception. The list of parser exceptions may be a superset of the cur-

rent P4 list which is given in Section **??**:

Identifier	Exception Event
p4_pe_index_out_of_bounds	A header stack array index exceeded the declared bound.
p4_pe_out_of_packet	There were not enough bytes in the packet to complete an extraction operation.
p4_pe_header_too_long	A calculated header length exceeded the declared maximum value.
p4_pe_header_too_short	A calculated header length was less than the minimum length of the fixed length portion of the header.
p4_pe_unhandled_select	A select statement had no default specified but the expression value was not in the case list.
p4_pe_data_overwritten	A given header instance was extracted multiple times.
p4_pe_checksum	A checksum error was detected.
p4_pe_default	This is not an exception itself, but allows the programmer to define a handler to specify the default behavior if no handler for the condition exists.

Table 2: Standard Parser Exceptions

1.5.3 Stateful Objects

Counters, meters and registers maintain state for longer than one packet. Together they are called stateful memories. These are described in Section **??**. They are accessed via respective extern object types in the standard library. Generic method calls on these objects replace the earlier custom P4 syntax.

1.5.4 Checksums and Calculations

Checksums and hash value generators are examples of functions that operate on a stream of bytes from a packet to produce an integer. These are described in Section **??**. They are accessed via respective extern object types in the standard library. Generic method calls on these objects replace the earlier custom P4 syntax.

1.5.5 Action profiles

In some instances, action parameter values are not specific to a match entry but could be shared between different entries. Some tables might even want to share the same set of action parameter values. This can be expressed in P4 with action profiles. These are described in Section **??**. They are accessed via an extern object type in the standard library. Generic method calls on these objects replace the earlier custom P4 syntax. Action profiles are an example of a table modifier extern object type.

1.5.6 Digests

Digests serve as a generic mechanism to send data from the middle of a P4 block to an external non-P4 receiver. This receiver can be anything from a fixed-function piece of hardware to a control-plane function. The *generate_digest* primitive action is described in Section **??**. This is accessed via an extern object type in the standard library. A generic method call on such objects replaces the earlier custom P4 action.

1.6 Standard Switch Architecture

The Standard Switch Architecture defines a highly abstract packet forwarding architecture geared towards packet switching. It serves as:

- An example P4 target architecture specification; and
- A widely supported architecture for simple yet portable P4 programs

While this architecture is designed primarily to allow the expression of packet switching programs, it is flexible enough to implement more advanced behavior. Other simple architectures geared towards different environments, such as NICs, could also be defined. The architecture is described in Section **??**. As for all targets, there is an associated Standard Switch Library, containing extern type objects.

1.6.1 Programmable regions

The Standard Switch Architecture has three P4-programmable regions: parser, ingress, and egress. It provides prototypes for these. Note that this gives a more explicit meaning to the blocks declared in traditional P4 programs. A draft form of the intrinsic metadata associated with the various interfaces to these regions is given next, to give more detail on how this works. The metadata is defined using standard P4 *header_type* ojects.

1.6.2 Intrinsic Metadata

All three blocks receive a read-only metadata header containing basic information about the packet:

```
header_type packet_metadata_t {
    fields {
        bit<16> ingress_port; // The port on which the packet arrived.
        bit<16> length;       // The number of bytes in the packet.
                              // For Ethernet, does not include the CRC.
                              // Cannot be used if the switch is in
                              // 'cut-through' mode.
        bit<8>  type;         // Represents the type of instance of
                              // the packet:
                              //   - PACKET_TYPE_NORMAL
                              //   - PACKET_TYPE_INGRESS_CLONE
                              //   - PACKET_TYPE_EGRESS_CLONE
                              //   - PACKET_TYPE_RECIRCULATED
                              // Specific compilers will provide macros
                              // to give the above identifiers the
                              // appropriate values
    }
}
```

The ingress block also receives the exit result of the parser:

```
header_type parser_status_t {
    fields {
        bit<16> return_code;       // The final status of the parser.
                                   // 0 if parser returned 'accept'
                                   // TODO: Define other values
        bit<8>  user_error_data;   // An opaque value written by
                                   // user-defined parser exceptions
    }
}
```

The ingress block's output intrinsic metadata controls how the packet will be forwarded, and possibly replicated:

```
header_type ingress_pipe_controls_t {
    fields {
        bit<16> egress_spec;   // Specification of an egress.
```

```
                                // This is the 'intended' egress as
                                // opposed to the committed physical
                                // port(s).
                                //
                                // May be a physical port, a logical
                                // interface (such as a tunnel, a LAG,
                                // a route, or a VLAN flood group) or
                                // a multicast group.
        bit     drop;           // Do not send the packet on to the
                                // queueing system. Other functions
                                // like copy-to-cpu and clone will
                                // still occur.
        bit     copy_to_cpu;    // Send a copy of the packet to the
                                // slow path.
        bit<8>  cpu_code;       // Opaque identifier packaged with
                                // the packet, when sending to the
                                // slow path.
    }
}
```

The egress block receives further read-only information about the packet determined while it was in the queueing system:

```
header_type egress_aux_packet_metadata_t {
    fields {
        bit<16> egress_port;      // The physical port to which this
                                  // packet instance is committed.
        bit<16> egress_instance;  // An opaque identifier differentiating
                                  // instances of a replicated packet.
    }
}
```

The egress block's output intrinsic metadata no longer has access to the egress spec for writing, since the packet has already been committed to a physical port:

```
header_type egress_pipe_controls_t {
    fields {
        bit     drop;           // Do not send the packet out of its
                                // egress port. Other functions
                                // like copy-to-cpu and clone will
                                // still occur.
        bit     copy_to_cpu;    // Send a copy of the packet to the
```

```
                                // slow path.
        bit<8>  cpu_code;       // Opaque identifier packaged with
                                // the packet, when sending to the
                                // slow path.
        bit     recirculate     // If true, recirculate packet to
                                // ingress parser
    }
}
```

1.6.3 Egress Port Selection, Replication and Queuing

The Standard Switch Architecture's egress mechanism is as described in Section **??**. This is a mechanism that is provided by this particular architecture, rather than something inherent to P4.

1.6.4 Cloning, Mirroring, Resubmission and Recirculation

The Standard Switch Architecture's cloning, mirroring, and resubmission and recirculation mechanism are as described in Section **??**. These involve extern object types that are provided by the associated Standard Switch library, rather than actions inherent to P4.