Proposal for serializable types in P4

P4 Serializable Types Subcommittee

Abstract

The current $P4_{16}$ specification includes descriptions of how data can be extracted from inconming packet data (through `packet_in`) in a parser and how it can be emitted as packet (through `packet_out`) in a deparer. It also identifies how arbitrary data may be packed in a tuple for communicating with an extern. In at least some casess, there is an implicit assumption that data communicated to externs through a tuple in this way will be handed to the extern as a packed data structure with no padding (or that the extern will undertand how to remove this packing). In this document we refer to the first type as data having an "emitable" type and he second as data having a "serializable" type.

In both cases these are described through the externs that interact with the data, rather describing these as general types that could be used to describe interactions with general externs. This proposal attempts to address this by explicitly identifying and describing "emitable" and "serlizable" behavior.

In addition, we propose extending the enumeration type to allow a backing type and representation to be specified for use in extraction, emission, and serialization.

Furthermore, propose that "serializable" types can be cast to and from appropriately sized bit-vectors.

The latter parts of this proposal have come up as requests to the P4 working group and it makes some sense to incorporate them here.

1. The need to specify emit- and serialize-ability

P4, as a language, is largely about reading incoming data in an wire-format form, parsing and processing that data, and then emitting this data out on the wire. In addition to parsing binary data into P4 header types and emitting those headers back out into binrary data, several externs make use of raw (or perhaps better stated, re-shaped) data to compute their final values. A good example of the later is the IP checksum computation which effectively re-interprets the

fields of an IPv4 header as a set of 16-bit values to be processed. In addition, casting from structured to unstructured data([1], [2]).

The parsing/emitting class of interactions is refered to as emitable data, while the arbitrary data passed as tuples to externs or casts between structured and unstructured data is refered to as serializable data. The difference between serializable and emitable data is focused on what happens with the header validity data on headers, header stacks, and header unions. In the case of emitable data, the header validity information might be set as part of an extract-like operation and it might be used to determine whether an emit-like operation emits any data, however, the flag itself is not expected to be read or written directly to or from the raw data. This is really just re-iterating what is already described in the `packet_in` and `packet_out` descriptions. Serializable data, on the other hand is expected to pack all data. This includes validity information, if we decide `header`s should be serializable. In both cases, the raw data is expected to be a packed-bit vector with a layout matching the order of fields in the structured data.

Currently, there are a handful of emitable types (it is worth noting that `packet_in.extract` and `packet_out.emit` do not completely agree on this set, we've used the more conservitive emit set here): headers, header stacks, header unions, and structs that are ultimately `header`s, header stacks, or `header_union`s at the leaves, e.g. a `struct` containing one or more `struct`s that contain hearders is okay, but a struct that contains a raw `bit<w>`, `int<w>`, `varbit<w>`, `enum`, or `boolean` is not emitable. Similarly, we can see examples in the P4$_{16}$ specification of serializable types: `bit<w>`, `int<w>`, and tuples containing serializable types, and `bool` values can be explicitly cast to a `bit<1>` representation, and thus is arguably also serializable.

In addition to codifying the behavior that arguably already exists in the P4$_{16}$ specification, we would also propose extending the concept of serializability to casts, such that non-bit vector data can be treated as bit-vector data and appropriately sized bit-vector data can be cast to structured data.

We also propose defining serializable (e.g. packed bit layout) formats for enums, booleans, varbits, headers, and header unions, and structures that contain these types or other structures that meet this criteria.

The proposal includes additional syntax for emitable and serializable types, instead of relying on the annotations currently available as part of P4$_{16}$. Part of the reason for this is to formally include this in the language, to avoid the possibility that an implementation might ignore the annotations leading to either different than expected semantics, or surprising error messages from the com-

[1] Issue 383: P4_16: Allow 'bit-vector structs' to be used everywhere that bit type can
[2] Issue 342: P4_16: Allow headers to contain 'bit-vector structs'

piler.

2. Adding emitable and serializable type modifiers

When specifying concrete types, it is not necessary for the programmer to specify that a type is emitable or serializable, since the compiler can determine this, however, when declaring types, the emitable and serializable modifiers can make explicit to the compiler that an emitable or serializable type must be supplied for the extern. Comment: We could also see this as being part of a type-generic hierarchy Comment: barefoot is considering for addition to the language. If that were Comment: the case then there might not be a reason for including explicit Comment: syntax for this purpose.

For example, the current definition for the `packet_out` extern is:

```
extern packet_out {
    void emit<T>(in T data);
}
```

The description types that can occupy `T`, includes `header`, `header_stack`, `struct`, and `header_union` types. However, `emit` is limited to only emitting `struct` whose fields are `header`, `header_stack`, or `header_union` types or fields that are of the same limited `struct` type.

This creates an implicit differentiation between those structs that are emitable and those that are not, at least in the case of `emit`.

We would propose that `emit` would make use of the `emitable` type modifier to specify that there is an expectation of emitable types here:

```
extern packet_out {
    void emit<T>(in emitable T data);
}
```

An example of the `serializable` from `checksum1-bmv2.p4` example program in the `p4c`:

```
verify_checksum (true, {
  hdr.ipv4.version, hdr.ipv4.ihl, hdr.ipv4.diffserv,
  hdr.ipv4.totalLen, hdr.ipv4.identification, hdr.ipv4.flags,
  hdr.ipv4.fragOffset, hdr.ipv4.ttl, hdr.ipv4.protocol,
  hdr.ipv4.srcAddr, hdr.ipv4.dstAddr, hdr.ipv4.options},
```

```
  hdr.ipv4.hdrChecksum, HashAlgorithm.csum16);
```

Here the tuple data is expected to be passed on raw to the `verify_checksum` or at least the underlying implementation is expected to be aware of any padding that might occur and compensate for it. The contents of the tuple must all be serializable as well, `bool` or `enum` values cannot be represented here, for instance.

The `verify_checksum` type in the `v1model.p4` include file is:

```
extern void verify_checksum<T, O>(in bool condition, in T data,
              inout O checksum, HashAlgorithm algo);
```

We would propose restricting the incoming data type to use the `serializable` type here:

```
extern void verify_checksum<T, O>(in bool condition,
              in serializable T data, inout O checksum,
              HashAlgorithm algo);
```

2.1. Casts

TBD

2.2. Marking `emitable` and `serializable` types

P4 programmers can make use of `serializable` for specifying when a concrete type should be checked to be serializable. For instance, a struct might be marked serializable to ensure that its components are all serializable. The compiler would be expected to raise an error (or maybe a warning?) when the type violates this serialize-ability.

```
header h1 {
  bit<8> f1;
}

header h2 {
  bit<16> f2;
}
```

```
emitable struct s1 {
  h1 f3;
  h2 f4[10];
}
```

3. Extending other types to be serializable

Currently, some scalar types (like `bool` types) are not explicitly serializable unless they are included as part of a `header` in the current spec, additionally, some types like `enum` cannot be serialized even though there are some interactions, such as those with the control plane, where the programmer may want control over this.

3.1. Extending scalar types

The integer types `bit` and `int` are inherently serializable, and unsurprisingly make up the fields of a `header`. We propose treating these as serializable when they appear as a stand alone value or as a field in a struct, which externs like `emit` currently do not support. Similarly, boolean values should be serialized as `0` for `false` and `1` for `true` and be backed by a `bit<1>`, following the explicit cast from `bool` to `bit<1>` in P4.

3.2. Making serializable enums

P4's `enum` type is currently left up to the compiler to determine the representation and backing type for the enum, however, as the current P4 spec acknowledges, there are instances where an enum is exposed to the control plane, and a compiler must specify how it will handle such enumerations in communication with the control plane. Instead of having this be a place where one implementation might vary from another, we propose that the `enum` type be extended to allow specifying both the backing type and the numeric representation of enumeration items.

We propose an extending `enum` as follows:

```
enumDeclaration
    : optAnnotations ENUM name '{' identifierList '}'
    | optAnnotations ENUM typeRef '{' specifiedIdentifierList '}'
```

```
specifiedIdentifierList
    : specifiedIdentifier
    | specifiedIdentifierList ', ' specifiedIdentifier

specifiedIdentifier
    : name '=' initializer
```

This keeps the original `enum` for use internal to the P4 program, but also provides a serializable `enum` with both the backing type and the numeric representation specified. In this case, we expect the `initializer` to be a compile-time known values.

4. Serialization of data

The $P4_{16}$ specification for `emit` defines how `header`, `header_stack`, `header_union`, and `struct` types should be handled for `emit`. This specification should be extended to include serializing `struct` fields that are scalars and serializable enums.

5. A note about implementation

Serializable types need not be represented as in their serializable format internal to the data plane. The serializable layout is only required for reading from the wire or writing to the wire, or in other extern functions or methods that specify a `serializable` modifier.

5.1. Why might the compiler decide on a different layout?

In some cases the serialized layout might impact performance when referencing or updating data. For instance, on a CPU, extracting bit fields, may require shifting and masking to extract a field. In cases where the field is accessed frequently, it might make more sense to choose an internal representation that is more efficient for referencing the field, and then do the work necessary to serialize this when the serialized layout is needed.

It is up to the compiler to determine what the representation should be, the serializable indicator only requires that when the value is passed to an extern function or method requiring the serialized data that it be serialized.