Proposal for serializable types in P4

P4 Serializable Types Subcommittee

Abstract

The current $P4_{16}$ specification includes descriptions of how structured data can be extracted from incoming raw packet data (through `packet_in`) in a parser and how it can be emitted as raw packet data (through `packet_out`) in a deparser. It also provides a number of entities, such as, externs, tables, and interactions with the control plane, that require communication of data in an expected layout. The data moving across these boundaries is limited to data types that are either scalar types with specified bit widths and representations, or aggregate types that contain these types. We refer to these data types as "serializable", and propose that this property of the type should be explicitly specifiable in P4.

In addition, we propose extending the enumeration type to allow specification of a backing type and representation, allowing such enums to be used as serializable data. We also recommend allowing a cast operation between a serializable type and an appropriately sized `bit<w>` type. These features have been requested from the P4 community in the past.

1. The need to specify serializability

P4, as a language, is largely about parsing incoming data from wire-format form into structured data, processing that data via match action tables, and then deparsing structured data back to wire-format to emit it on the wire. In addition to parsing binary data into P4 header types and emitting those headers back out into binary 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, specified through a tuple, as a set of 16-bit values to be processed.

The data types that can interact with these behaviors is limited to scalar types with specified bit widths and representations and aggregate types that contain these types. We refer to this as "serializable" data and propose that it should be possible to explicitly specify when a serializable type is required

or when a user specified type is expected to be serializable. We can imagine using either adding a `serializable` type specifier, or building this idea into the proposed typed hierarchy for generics.

It is worth noting that not all serializable types are handled in the same way. The `packet_in.extract` method must set a header validity flag when data is extracted into that header. The `packet_out.emit` method is currently limited to operating on header types or types that contain header types, and the `emit` method uses the validity information to determine if a header should be written to the wire but does not write the validity information. We would like to see `packet_out.emit` extended to match the types supported by the `packet_in` methods, which allow for other serializable types to be extracted.[3]

Where serializable data is used elsewhere, such as tuples passed in to an extern like the IP checksum extern, we expect that data to be packed-bit data, without any padding. In particular, tuples and structs with multiple fields are expected to be read and written without any padding bits. In the case of headers, if they are treated as serializable types they will need to read and write the validity information along with the header data.

Specifying the serialized layout also allows aggregate data to be cast between a structured data type and a `bit<w>` field, as has been requested in P4 spec issues.[1] It is worth noting that this type of cast is already possible in the language through the use of shift and or operations (to convert from structured data to a `bit<w>`) or bit slice operations (to convert from a `bit<w>` to structured data).

In addition, we propose extending the syntax of the `enum` type to allow a backing-type to be specified along with numeric values for the entries in the `enum`.[2] This would allow for serializable `enum`s to be used as fields in headers and provide a way for the programmer to specify the `enum` representation.[4]

It is worth noting that none of these recommendations are intended to force implementers to use a particular representation for data. A CPU target, for instance, might want to specify aggregate data types with padding to better match the memory model on the CPU. Or an implementation might decide to completely flatten and re-arrange data to better fit a bus between parts of a hardware device. The intention is to provide some flexibility to the P4 programmer.

[3] Effectively, this re-opens the discussion from Issue 161: Types supported by emit

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

[2] Issue 550: Allow enums with specified bitwidths and values Issue 394: Allow enum to 'derive' bitstring type

[4] This addresses the note in section 8.3 of the P4 spec for `enum`s that appear in the control-plane API.

In either case, a cast or other bit packing or bit extraction operation might have some cost, an where this cost exists it is recommended that the implementation include this in documentation for the implementation so that the end user is aware of the cost model.

2. Adding serializable type modifiers

When specifying concrete types, it is not necessary for the programmer to specify that a type is serializable, since the compiler can determine this, however, when declaring types, the `serializable` modifier can inform the compiler that the type should be checked for this property. The `serializable` modifier is more useful in the case where an extern, control, or other callable P4 element requires a parameterized type be serializable. This idea dove tails well with the idea of specifying constrained generics, and if it is captured in the type hierarchy, the need for a separate `serializable` modifier.

2.1. Marking arguments as `serializable`

For example the `verify_checksum` call from the `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);
```

Alternatively, we can imagine specifying the type with the generics (apologies to Nate and the team working on this if I've got the syntax incorrect):

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

We would propose that the `packet_in` and `packet_out` externs be similarly annotated with the `serializable` modifier (or possibly an `emitable` modifier see note below).

Note. The question around whether "emitable" is a behavior on serializable data or a data type that is a super set of `serializable` data is something I've been struggling a bit with, and something I think we should consider carefully.

The argument for "emitable" as a behavior is that we think of `header` data as being serializable with the valid bit included when a `header` is serialized, and the valid bit not included when a `header` is emitted. I think the downside of this is that the type system in P4 does not capture this kind of "effect" information in the type, and I think there is potential for this to be confusing to end users because it is sort of expressed on an orthogonal direction to how we think about serializable or not serializable.

The argument for "emitable" as a type, is that we can think of serializable data as being `bit<w>`, `int<w>`, and `struct` types containing these types, and `emitable` data as `header`, `header` stack, `header_union`, and `struct` types that contain these types. This makes it somewhat clearer in the type system where the lines are drawn, because "emitable" data is a super set of serializable data, with serializable data simply being treated as "always" valid when used where an "emitable" type is expected. The downside of this approach is that `header` and `header_union` types are harder to treat as "serializable" since, the extra validity information has to go somewhere.

The description below treats emitable as a type while the rest of the document discusses it as a behavior, reflecting my own questions about how this should be handled.

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 `struct` types with fields of these types.

This creates an implicit differentiation between `struct` types 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);
}
```

Or, again, following from the generics proposal:

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

2.2. Marking `serializable` aggregate 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 when the type violates this property.

For instance, the following `struct` would be checked to ensure its fields are serializable.

```
serializable struct s1 {
    bit<10> a;
    bit<2>  b;
}
```

Note. I'm not sure how the generics example is going to differentiate serializable and non-serializable `struct`s, but we would presumably follow suite.

So far, so good, but the compiler could easily determine this serializability itself. What happens if a `struct` includes another `struct`:

```
serializable struct s2 {
    bit<8> a;
    s3     s;
}
```

If `s3` is serializable, then the compiler has no reason to complain.

Note. Whether `s3` itself needs to be marked as serializable, or if it is sufficient for the compiler check the structure of `s3` and determine it is serializable, is something we should decide.

However, if `s3` is defined as follows:

```
struct s3 {
    error recordedError;
}
```

The compiler would be expected to mark `s2` as an error, because it cannot live up to the requirement of being serializable.

3. Casts

One of the requests we have heard from the community[1] is that they would like a way to treat structured data as a `bit<w>` type or to treat `bit<w>` type as structured data. We would propose that data specified as serializable can be cast to a `bit<w>` type and that a `bit<w>` type can be cast to `struct` data.

Following from the expectations that $P4_{16}$ outlines around avoiding ambiguous casts (such as those that simultaneously change signed-ness and extend bit width), we recommend that the width specified by the bit width must match exactly the bit width of the serializable type when casting between a `bit<w>` and `serializable` type.

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

For instance, in the following code fragment we cast back and forth between a `serializable` type and an appropriately sized `bit<w>`.

```
serializable struct s_t {
    bit<7>  a;
    bit<10> b;
    bit<7>  c;
}

s_t     s;
bit<24> raw;

s   = (s_t)raw;
raw = (bit<24>)s;
```

However, the following casts would not be allowed:

```
bit<32> raw2;
int<24> raw3;

raw2 = (bit<32>)s;
raw3 = (int<32>)s;
```

In these cases the program would need to supply multiple casts. Part of the reason for this is to make sure that it is clear to a P4 programmer what is being done.

It is worth noting that it is currently possible to support this in P4 through P4 operations:

```
raw = (s.a << 17) | (bit<24>)(s.b << 7) | (bit<24>)(s.c);
s.a = raw[17,23];
s.b = raw[7,16];
s.c = raw[0,6];
```

We also propose that if a user wants to cast between two `struct` types that are the same size, but are not otherwise structurally similar, they must first cast through a `bit<w>` type. Again, this is to encourage the program writer to be aware of the translations happening, even if the compiler can effectively compile away the operations.

4. Making serializable `enum`s

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.

Another approach, would be to use annotations for specifying the backing type and numeric values for the type. The upside of this approach is that it does not require changing the syntax of P4 to support the newer representation. The downside is that backends that ignore these extensions will not handle these `enum`s as serializable. In the case where an `enum` would be used in a place a serializable type would be used, this will result in an error, however, in cases where it will might be exposed to the control plane, and the compiler might have already had its own method for dealing with `enum` representations, it will silently differ from the users expectations. In either case, I would argue, the work needed on the part of implementers is similar, and extending the language is arguably both safer and backwards compatible, since existing programs will continue to work exactly as they have.

5. Extending other types to be serializable

Currently, some scalar types (like `bool` and `varbit` types) are not explicitly serializable unless they are included as part of a `header` in the current spec, additionally, some types like `header` and `header_union` have a specified handling through `extract` or `emit` but not a clear serializable layout that would capture their validity information.

5.1. Extending emit behavior to scalar types

The integer types `bit` and `int` are inherently serializable, and unsurprisingly make up the fields of a `header`. They can also be specified as `packet_in.look_ahead` (and possibly `packet_in.extract`) can be used to extract non-`header` types. However, `packet_out.emit` currently avoids handling these types.

We propose treating these as emitable 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.

It is worth noting that the reason given for the current restrictions on these are to avoid creation of non-byte aligned data on egress in implementations that restrict `header` types to be byte aligned. We would suggest that implementations with this restriction could reject programs with non-byte sized scalar types, in lieu of doing more expensive static analysis or run time checking to enforce this restriction. This opens the door for more flexibility even in those implementations with this restriction to handle more flexible deparsers.

5.2. Extending `header` types

We can imagine extending the concept of general serializability to `header`, `header_union`, and `varbit` types, by specifying how validity data (in the case of `header` and `header_union`) and size data (in the case of `varbit`) data would be serialized.

The `header` type is the simplest to handle, with a single additional bit to represent the validity of the header. This bit could be supplied following the header data.

The `header_union` type might have a more compact representation with an additional bit field with enough additional values to specify which `header`, if any, is currently valid. For instance, a `header_union` over three `header` types would have 4 valid representations: no headers, `header` 1 valid, `header` 2 valid, or `header` 3 valid, and could be represented by a `bit<2>` or an equivalent `enum` with a `bit<2>` backing type.

The `varbit` would need both enough space to represent the largest possible type, along with a dynamic size field, sized to the possible sizes of the `varbit`.

5.3. Make `bool` types serializable

We can also imagine making the `bool` type serializable and using its current `bit<1>` cast behavior to define its serializable behavior. This would make it obvious how a `bool` within a header should be handled and encode into the type how it is expected to be used.

6. Implications of serializable types

In addition to making explicit what types are serializable (and emitable) and how this data would be handled independent of a particular extern, we imagine that serializable types could lead to P4 programmers being able to create useful type abstractions.

For instance, we foresee serializable `enum` and `bool` types being used in `header` definitions to allow for use symbolic names for types and matches in the transition select statement.

We can also imagine serializable `struct` fields being used in a `header` to allow for `header`s that have shared structure to express this shared structure more succinctly. This is currently possible through use of the C preprocessor, but we find this a bit of a clumsy way to handle this.

7. 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.

7.1. 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.