Proposal for serializable types in P4

P4 Serializable Types Subcommittee

Abstract

The current $P4_{16}$ specification does not explicitly call out which types are serializable and which types are not, instead attempting to describe in the description of certain features how various types are handled when they are written to the wire. This has both lead to some confusion and also meant that there is not a consistent way to describe serializable types and interfaces which require serializable types. In addition to these deficiencies, certain types, like enums, do not have a specified type or serializable data representation, even though the specification acknowledges that a representation must be specified for interacting with the control plane.

We propose that an explicit definition of what is and is not serializable should be specified, and types like enums have the option of having a serializable representation when it is required.

1. The need to specify serialize-ability

P4, as a language, is largely about reading incoming data in a serialized (wire-format) form, parsing and processing that data, and then serializing this data to put it back on the wire. In addition to this obvious use of serializable types, communication with the control plane and parser/deparser stages within the pipe line (potentially internal to the chip or fabric, as described in the PSA) may also have the need to serialize data from one part of the pipeline to communicate it with another section of the pipeline.

Up until now, P4 has not made an explicit distinction between serializable and non-serializable types, instead using the `header` type as the source of serialize-ability, but then extending this in some cases (like `emit`) to include some subset of the `struct` type.

Additionally, P4 has not provided a way to make certain types serializable, like enums, which might be communicated within a P4 program through serialization between pipeline stages, or communicated with the control plane, as has been noted in the spec. It would be useful to allow the programmer to

specify how enums that the programmer knows will need to be serialized with both a backing type and the values for the enum elements.

2. Adding a serializable type modifier

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 make it explicit to the compiler that a serializable type must be supplied for the extern.

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 serializable and those that are not, at least in the case of `emit`.

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

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

P4 programmers can also 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;
}

serializable 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.