Pollux: Protobuf Compatibility Checking

Matt Schwennesen

December 31, 1979

Contents

1 Introduction

Software has become an integral and integrated part of everyday life, and with that comes a near constant stream of updates to various devices all around us. Unfortunately, software updates are a frequent source of issues [4, 1], often making it into production environments before the necessary edge cases are triggered. We believe that techniques from formal verification can be applied here to ensure update compatibility, which is ultimately what my project aims to do. Unlike some previous work [2, 3], I aim to prove compatibility of software updates without imposing restrictions about how the update itself is performed.

The most nebulous part of this goal is how to define "compatible", which I break down into two major categories; data compatibility and operational compatibility. Data compatibility aims to show that that the new version can correctly interpret persistent state left behind by the pervious version while operational compatibility reasons about behavior of invoking the same function or feature in both versions of the software. This project is currently working on data compatibility.

Imagine that you're a developer for a distributed system using `protobuf` and `grpc` to communicate between nodes. A new version of `protobuf` just came out and added a syntax for maps to the protocol. When you used to need maps, you had to manually encode a list of key-value pairs but now you can write `map<string, int32>` in the `.proto` schema file. Is changing between these syntactic definitions a compatible update? From a high level, they both model the same mathematical map, which is the information the program is actually trying to communicate, so it should be possible to define a compatible update this way. The specific answer naturally depends on how the new `map` is encoded into binary, and in the case of `protobuf` a `map` is syntactic sugar for a list of key-value pairs so this is a valid update [1]. While this example is considering a rolling update to a distributed

[1] There is slightly more to it than that, regarding some `optional` tags which might not present in the original definition but it is possible to define this update in a compatible way

system, the same question would apply to an application which writes the serialized blob to disk and reads it when the application starts up again.

The notion of both types of maps encoding the same mathematical map leads us to the idea of abstract state. When we're programming, we don't often think about which bucket an element of a hashmap is in but rather we think at a higher level. It is possible to conceptualize this formally, were we have some abstract state and a concrete state which are linked with a predicate asserting that the abstract state is captured by the concrete state. This is the same relationship we see in proofs of functional correctness, were the specification of a function corresponds to the abstract state and the implementation to the concrete state. This relationship will serve as the theoretical backing to the project.

More practically, consider an RPC or other serialization library such as Protobuf or Cap'n'Proto. Updates to RPC schema are an excellent place to start exploring notions of data compatibility since RPC schema are separated from the source code that operates on them, are highly structured serialized formats and are designed to allow the schema to evolve though updates. Tools already exist which can report on if two versions of a schema file are compatible, these tools operate using ad-hoc notions of compatibility asserting a specification or verifying the implementation. Using F*, I plan to verify a compatibility checker for one of these libraries before expanding the project to consider a wider range of data formats. It should also be possible to include some analysis of the source code operating on the state to prove more invasive updates, or even the proofs of verified pieces of software, although that is a goal for a longer timescale.

2 Abstract State Schema

In software verification, an implementation is always verified against some specification of correct behavior. This is implicitly part of `protobuf` too; when a message is serialized and transmitted over the network, the developer expects that the same structure can be recreated by the receiving program. Refactoring this intuition to be more directly applicable to the standard verification setup, one could argue that the binary wire blob serves as an "implementation" of the structure or object in the host language (be it `go`, `java`, `python` or any other language with `protobuf` bindings) which itself is generated by `protoc` and is an "implementation" of the proto schema defined in a `.proto` file. This stack defines the `.proto` file as the top level source of truth, but this can be modeled by the high level host language in most cases. From the prospective of an application developer, the host language representation is the most important, since this is where the schema structure interacts with other parts of the host application.

The definition of a `proto` message can be modeled by an F* type, in a way similar to structures created by `protoc` for other host languages. This is not fully sufficient though, since the representation exposed by `protoc` isn't enough to fully reason about compatibility.

2.1 F* Proto Representation

Some parts of the translation from a `proto` message [6] into F* is simple. For example, a 64 bit unsigned integer can be represented with a `UInt64` from the F* machine integer library. Likewise a message can be represented by a record [5]. Not everything cleanly generalizes though. Specifically, it is unclear to how represent some primitive types such as `float`, `double` or `string`.

`proto` Feature	F* Representation
Integers	Machine Integer Library
`string`	`string`
`message`	Record
`optional`	`option`
`repeated`	`list`
`map`	???
`bool`	`bool`
`bytes`	`Seq.seq UInt8.t`
`enum`	Inductive type
`oneof`	Inductive type (Sum type)
`option`	???

Table 1: `proto` language features and corresponding F* representations.

Notice that there are several unknown representations in the table. Below is some commentary on these features:

- `map`: As far as I can tell, F* doesn't have maps, however this F* development doesn't *need* maps beyond a way to serialize and de-serialize them. We should be able to develop a parametric `map` definition as a list of tuples, just like how protobuf would serialize them.
- `option`: This isn't an optional value, but rather top level options used to set things like the `java` package. At the moment, I'm not sure if or how changing values can impact protobuf compatibility.

A good completeness test might be encoding `descriptor.proto`, the proto file which can encode other protobuf schemas.

While this would provide a framework for building F* bindings for protobuf, a richer representation for a protobuf schema for checking compatibility at the wire level.

2.2 Compatibility Checking with F* Types

When a protobuf message is encoding, field names are stripped out in favor of field numbers or field tags, which are manually defined in the protobuf schema. Consider the two `proto` snippets below:

```
message Foo {
    int32 bar = 1;
}
```

```
message Foo {
    int32 bar = 2;
}
```

Figure 1: Example showing the importance of including proto field numbers in compatibility checking.

Any invocation of `protoc` will generate exactly the same application-facing code, even though a message generated using the left proto file will be parsed to an empty message by the right proto file.

At the moment, I'm not sure of the exact way to represent the field tag in the F* type. There are probably multiple ways to do this, such as making everything a record or tuple with an integer or placing the fields into some generic wrapper which includes the field tag. This feels monadic, but I'm not sure about the details right now.

3 Protobuf Compatibility Questions

As the F* development moves forward, it is beneficial to have a minimal working example of a non-trivial compatibility question for protocol buffers. There are several clearly trivial questions, like updating a `int32` to an `int64` using the variable width integer encoding.

After thinking about this for a while (very scientific), this was the simplest example that I wasn't immediately sure of the outcome:

```
message Bar {
    string baz = 1;
}

message Foo {
    Bar bar = 1;
}
```

```
message Foo {
    string bar = 1;
}
```

Figure 2: A compatibility question.

On second thought, the data is correctly exposed in the string on the right but there will be several junk bytes at the beginning that the application has to trim. As a definitional question, should this be considered compatible?

References

[1] Jim Gray. "Why Do Computers Stop and What Can Be Done about It?" In: *Symposium on Reliability in Distributed Software and Database Systems*. 1986.

[2] Sameer Ajmani, Barbara Liskov, and Liuba Shrira. "Modular Software Upgrades for Distributed Systems". In: *ECOOP 2006 Object-Oriented Programming*. Ed. by Dave Thomas. Berlin, Heidelberg: Springer, 2006, pp. 452–476. ISBN: 978-3-540-35727-8. DOI: 10.1007/11785477_26.

[3] Mark Reitblatt et al. "Abstractions for Network Update". In: *SIGCOMM Comput. Commun. Rev.* 42.4 (Aug. 13, 2012), pp. 323–334. ISSN: 0146-4833. DOI: 10.1145/2377677.2377748. URL: https://doi.org/10.1145/2377677.2377748 (visited on 01/23/2025).

[4] Yongle Zhang et al. "Understanding and Detecting Software Upgrade Failures in Distributed Systems". In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. SOSP '21. New York, NY, USA: Association for Computing Machinery, Oct. 26, 2021, pp. 116–131. ISBN: 978-1-4503-8709-5. DOI: 10.1145/3477132.3483577. URL: https://dl.acm.org/doi/10.1145/3477132.3483577 (visited on 10/30/2024).

[5] Nikhil Swamy, Guido Martinez, and Aseem Rastogi. *Proof-Oriented Programming in F**. 2023.

[6] *Language Guide (Proto 3)*. URL: https://protobuf.dev/programming-guides/proto3/ (visited on 04/09/2025).