In order to reason about Protobuf compatibility, we need a way to represent both Protobuf values and descriptors. While the goal here isn't *technically* to implement a Protobuf serializer and parser in F*, that is a natural step on the why to a protobuf compatibility checker.

0.1 Representing Protobuf Values

The most natural way to represent a protobuf message value would be a record type [**swamy2023proof**], but there is no mechanism to construct these types dynamically in F*. Limitations like this in most languages with Protobuf implementations coupled with the near-perfect fit of using a record for a message likely informed the choose of having a Protobuf compiler (`protoc`) which outputs code in the host language. Since Pollux is not interested in being a `protoc` plugin or anything like that, the record representation is not suitable for our purposes. Rather, Pollux defines a Protobuf message value with this type

```
type vf = string & vty

let sort_vf (v1:vf) (v2:vf) : bool = (bool_of_compare String.compare) v1._1 v2._1
let msg_names (m:list vf) : list string = map fst m

type msg = m:list vf{List.sorted sort_vf m /\ List.noRepeats (msg_names m)}
```

Figure 1: The F* type Pollux uses to represent a message value, essentially a list of name value pairs, sorted by name with the additional requirement that all the names are unique.

In the `vf` type above, `vty` is a value of an individual field. At the moment, Pollux doesn't support nested messages, `oneof` fields, enums or `map` fields. However, all of the basic field types are supported. Table 1a shows how each of the scalar types as well as `string` and `bytes` types are represented with Pollux. Notice particularly for the integer types, how exactly the integer is encoded has been elided from the value representation. This is done intentionally, since those are encoding details that will not be visible to any protobuf host language.

```
type dvty (v:Type) =
| VIMPLICIT : v -> dvty v
| VOPTIONAL : option v -> dvty v
| VREPEATED : list v -> dvty v

type vty =
| VDOUBLE   : dvty double -> vty
| VFLOAT    : dvty float -> vty
| VINT      : dvty int -> vty
| VBOOL     : dvty bool -> vty
| VSTRING   : dvty string -> vty
| VBYTES    : dvty bytes -> vty
| VMSG      : dvty unit -> vty
| VENUM     : dvty unit -> vty
```

Figure 2: The F* types Pollux uses to represent values in a message. The two-layer inductive structure ensures that programs can find both the type of the value being stored and how it's modified (i.e. repeated or optional decorators).

proto	F⋆
Integers	`int` type
`string`	`string`
`message`	`msg` type
`bool`	`bool`
`bytes`	`list U8.t`
`double`	`list U8.t`
`float`	`list U8.t`
`optional`	`option`
`repeated`	`list`

(a) Since F⋆ lacks floating point types, Pollux merely stores them as a list of bytes refined to be the appropriate length.

proto	F⋆
`map<K, V>`	`list (K × V)`
`enum`	`s:string{Some? find ((=) s) d}`
`oneof`	`(s:string × vty){Some? find ((=) s) d}`
`message`	`msg` type
`option`	???

(b) `proto` language features not currently supported by Pollux, but listing my ideas about them

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

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

- `map`: While F⋆ does have maps, the construction is based on functions, like you might find in a λ-calculus. This makes it impossible to do things like iterate over all the keys or values of the map. It may be beneficial to develop a map implementation based on lists of tuples if these traditional map operations are required.
- `enum`: Just like using records to represent message values, it would make the most sense to use an inductive type to represent an enum. However, there is also no what to dynamically create an inductive type in F⋆. Instead, an enum can be represented as a string, specifically the name of the set value, that uses F⋆ refinement types to ensure that a valid option from the enum descriptor is set. It is worth noting that this potentially breaks from the established open enum behavior of proto 3 [**EnumBehavior**].
- `oneof`: Very similar to the `enum` case, except we require that the name of the set field is in the descriptor of the `oneof` field. The exact type written in Table 1b isn't correct as the `find` function will need to take the `fst` of the elements in `d` an it should probably include some refinement on the `vty` as well.
- `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 schema.

0.2 Representing Protobuf Descriptors

0.3 Compatibility Checking with F* Types

When a protobuf message is encoding, field names are stripped out in favor of field numbers and the type is reduced to a field tag, 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 3: 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. This is an example why just looking at a message value isn't sufficient to check the compatibility between two messages.

Instead, Pollux chooses to

`desc`	$: \mathit{Type}$	Type of a Protobuf descriptor
`msg`	$: \mathit{Type}$	Type of a Protobuf message value
`wf`	$: \texttt{desc} \to \texttt{msg} \to \mathit{Prop}$	well-formed proposition
$[\![\cdot]\!]$	$: \texttt{m} : \texttt{msg}\{\texttt{wf d m}\}$	A message well-formed w.r.t a descriptor
parse	$: (d : \texttt{Desc}) \to \texttt{Bytes} \to \mathit{option}\ [\![d]\!]$	Parse function
$\mathit{serialize}$	$: (d : \texttt{Desc}) \to [\![d]\!] \to \texttt{Bytes}$	Serialize function

Several shorthands arise from this, namely $\mathit{parse}_d : \texttt{Bytes} \to \mathit{option}\ [\![d]\!]$ for the partial application of a descriptor d to parse. The same thing can be done for $\mathit{serialize}_d : [\![d]\!] \to \texttt{Bytes}$, which naturally become more traditional definitions of a parser and serializer.