Despite being a "simple" format by the standard of many other real-world encoding formats, Protobuf still has a lot of features that need to be modeled, many of which are orthogonal to the challenges faced by formally verifying an implementation of the Protobuf format. To this end, consider a much simpler format designed to help establish the reasoning principles at play here.

0.1 Descriptor Definition

```
(* A field can be either an integer or a Boolean *)
Inductive FieldDesc : Set :=
| D_INT
| D_BOOL.

(* A descriptor is either a base field, or two nested descriptors. *)
Inductive Desc : Set :=
| D_BASE (n : string) (f : FieldDesc)
| D_NEST (d1 : Desc) (d2 : Desc).
```

This is intentionally a simple format, so that we can focus on working with nested descriptors. A descriptor describes the type we'll use for a message as well, defined by these two functions which compute a tuple type.

```
Definition FDescTy (f : FieldDesc) : Set :=
  match f with
  | D_INT => Z
  | D_BOOL => bool
  end.

(* A value corresponding to some descriptor is a large tuple, using names as
   strings and values as either ints or bools, respectively. *)
Fixpoint Denote (d : Desc) : Set :=
  match d with
  | D_BASE n f => string * FDescTy f
  | D_NEST d1 d2 => (prod (Denote d1) (Denote d2))
  end.
```

0.2 Encoding Format

While this is a simple descriptor setup, it is important to define what the binary format will be.

Each descriptor will be serialized as a tag for the type

- Base Descriptor ⇒ 0
- Nested Descriptor ⇒ 1

For the base descriptors, the next byte is another tag for the type of the field

- Integer field ⇒ 0
- Boolean field ⇒ 1

(Note: This could be removed if integer and Boolean fields both used the same [length] encoding)

The integer's will be encoded into a 4-byte little endian blob while the Booleans are encoded as 1-byte with 0 for false and any positive number for true.

Nested messages will be length prefixed with the number of bytes the rest of the message takes and then the encoded message.

0.3 Parsing & Serializing

An implicitly well-typed parser and serializer can be found on the Pollux GitHub as `src/SimplParse.v`. I will not be listing all the code here. In this case, implicitly well-typed means that we have the final parser type as `parse` $(d : Desc)$: `Parser` $[\![d]\!]$. This is considered implicitly well-typed since we know that the result of calling the resulting parser will always return a tuple constructed to exactly match the structure of the input descriptor.

We believe that we will need to move to an explicitly well-typed parser setup to scale to a format as complex as Protobuf. Under that setup, the type of the top-level parser would be `parse` $(d : Desc)$: `Parser Doc` for some generic, universal document model `Doc`. We'd also need to provide a theorem standing that "if `parse d enc` $\rightsquigarrow x$ then $\vdash_{\text{doc}} x : d$", basically that if calling `parse` on an encoding returns a result, the structure of that result will be well-typed under some document type system.

That is currently future work.

0.4 Correctness Proofs

In order to focus on the inductive nature of the final message proof, I will assume that the integer parser / serializer has been proven correct, as well as a number of the intermediate parser / serializer pairs.

Theorem 1 (Simple Parse Correctness)**.**

$$\forall\, x,\ d,\ enc,\ rest.\ wf\ x \rightarrow \texttt{SerialDesc}\ d\ x = \texttt{SerialSuccess}\ enc \rightarrow \texttt{ParseDesc}\ d\ (enc\ +\!+\ rest) = \texttt{ParseSuccess}\ x\ rest$$

Proof. We will proceed using induction on the descriptor. If a numeric measure is needed, that can be derived as the depth of the deepest nested descriptor. Consider first the base case there the input descriptor is `D_BASE`. There are two sub-cases to consider, one for each type of base descriptor.

- *Integer.* We unfold the serializer and match on the `D_INT` descriptor to reduce down to `SerialConcat SerialTags SerialZ4 (D_BASE n D_INT, (snd v))`. We know that the input value `v` is of type `Denote D_BASE n D_INT` or, equivalently, `String * ℤ`, `snd v` will be an integer. We can use the correctness theorem for the `SerialConcat` parser combinator (which is proved separately in the Rocq development).

□