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 descriptor is either a base field, or two nested descriptors. *)
Inductive Desc : Set :=
| D_BOOL
| D_INT
| D_NEST (d1 : Desc) (d2 : Desc).

Inductive Val : Set :=
| V_BOOL (b : bool) : Val
| V_INT (i : Z) : Val
| V_NEST (v1 : Val) (v2 : Val) : Val.
```

This is intentionally a simple format, so that we can focus on working with nested descriptors. However, it is possible to build a descriptor for inspecting only a value and we could write a function which initializing an "empty" descriptor given the close correspondence between them.

```
Fixpoint Schema (v : Val) : Desc :=
  match v with
  | V_BOOL _ => D_BOOL
  | V_INT _ => D_INT
  | V_NEST v1 v2 => D_NEST (Schema v1) (Schema v2)
  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 $\Rightarrow$ 0
- Nested Descriptor $\Rightarrow$ 1

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

- Integer field $\Rightarrow$ 0
- Boolean field $\Rightarrow$ 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

There has been some discussion about implicitly well-typed or explicitly well-typed parsers. In this case, implicitly well-typed means that we have the final parser type as `parse` $(d : Desc)$: `Parser` $[\![d]\!]$ while an explicitly well-typed parser setup would have the top-level type of `parse` $(d : Desc)$: `Parser Val` for some generic, universal document model `Val`. 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.

The current implementation is closer to being explicitly well-typed, as evident from the descriptor and value definitions above, however the typing theorem has not been formally stated or proved. Also notice that the current parser / serializer doesn't require a descriptor to parser or serializer any value, since all the required information is encoded in the value or the binary encoding itself.

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. The Boolean parser / serializer and the byte parser / serializer have been formally proved.
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, which is actually how the recursive combinator works.

First, we may assume that the value to be serialized is well-formed. The well-formed condition, wf, states that each integer embedded in a `Val` is within the range $0 \leq z < 2^{32}$ and that both "tags" for the `Val` fit within one byte, i.e. $0 \leq t < 256$.

Consider first the base case there the input `Val` is `V_INT z`. We unfold the serializer and are presented with a `SerialBind` combinator, which is designed to apply the outer tag (in this case 0) as a prefix to the payload. Continuing to unfold, we're presented with another `SerialBind` which applies the inner tag (also 0 in this case) before encoding the integer itself. Thus, we know that the encoding enc can be broken down as $0 :: 0 :: enc_z$ where we have two explicit tags and the integer encoding, learning along the way that $\texttt{SerialZ}_{32}\ z = \texttt{Success}\ enc_z$. On the parsing side, we first parse the outer tag, 0, then the inner tag to reduce the parser to $\texttt{Map ParseZ}_{32}\ (\lambda\ \texttt{z} \Rightarrow \texttt{V\_INT z})$ called on enc_z. By the correctness of the integer parser / serializer pair, $\texttt{ParseZ}_{32}$ will correctly parser `z` and the simple mapping function fully recovers `V_INT z`. Observe that we meet the well-formed condition for the integer parsers because the value well-formed condition also limits the range of integers to an appropriate value.

The second base case works the same, except it uses the Boolean parser / serializer rather than the integer one.

Now we come to the difficult case, the inductive case were $\texttt{v = V\_NEST v}_1\ \texttt{v}_2$. The parsing chain starts off the same has the base cases, with the serialization of the outer tag (which is 1), before proceeding to the inner tag, which is the length of the payload. By the well-formed condition, we

assume that this is less than 256, meaning it can be successfully encoded into one byte. In the inductive case, we also have access to two inductive hypotheses,

$$\forall\ enc,\ rest.wf\ v_1 \rightarrow \texttt{SerialVal}\ v_1 = \texttt{SerialSuccess}\ enc_1 \rightarrow \texttt{ParseVal}\ (enc_1\ +\!+\ rest) = \texttt{ParseSuccess}\ v_1\ rest \quad (\text{IH}_1)$$

$$\forall\ enc,\ rest.wf\ v_2 \rightarrow \texttt{SerialVal}\ v_2 = \texttt{SerialSuccess}\ enc_2 \rightarrow \texttt{ParseVal}\ (enc_2\ +\!+\ rest) = \texttt{ParseSuccess}\ v_2\ rest \quad (\text{IH}_2)$$

At this point, the serializer we're left with is `SerialConcat SerialVal SerialVal (v1, v2)`. (I'm currently eliding the fact that this process is actually happening in the `Recursive` serializer combinator, so the serializer we're dealing with takes a serializer which is presumed to behave like `SerialVal`). Either way, we know via assumption that the `SerialConcat` parser must succeed, since otherwise the original `SerialVal` call would fail, so by the inversion lemma on `SerialConcat`, we also know that `SerialVal v1` and `SerialVal v2` must succeed. Furthermore, there exists two encoding blobs such that

$$\texttt{SerialVal}\ \texttt{v}_1 = \texttt{Success}\ enc_1 \wedge \texttt{SerialVal}\ \texttt{v}_2 = \texttt{Success}\ enc_2 \wedge enc = enc_1 +\!+ enc_2$$

With these facts, we can apply each inductive hypothesis with the correct instantiation of *rest* to learn that

$$\texttt{ParseVal}\ enc_1 +\!+ (enc_2 +\!+ rest) = \texttt{Success}\ v_1\ (enc_2 +\!+ rest)$$

And

$$\texttt{ParseVal}\ enc_2 +\!+ rest = \texttt{Success}\ v_2\ rest$$

On the parsing side, we follow the same argument as before with the two tags. (Actually, the inner tag is used to length limit the parsing of the two nested values, although I think this isn't needed for this format. It will be required for others).

Similarly to the base cases, we reduce the parser, via the correctness of the matched `ParseBind` and `SerialBind`, down to `Map (ParseConcat ParseVal ParseVal) (λ vs ⇒ let (v1, v2) := vs in V_NEST v1 v2)`. Since `V_NEST v1 v2` is the input value we're trying to recover, it's clear that if the input parser to `Map` produces `(v1, v2)` the mapping function will produce a correct final result. Thus we only concern ourselves with the correctness of `ParseConcat ParseVal ParseVal (enc1 ++ enc2)`. Since we don't have the top-level theorem that `ParseOk ParseVal SerialVal` (which is what we're actively trying to prove), we cannot use the Concat correctness theorem. Instead, unfold the `ParseConcat` combinator, which starts with a call `ParseVal enc1 ++ (enc2 ++ rest)`. We already know this succeeds, producing a call to the right parser on the remaining data, `ParseVal enc2 ++ rest`. Using the result of the other inductive hypothesis, that also success and returns `(v1, v2)` as needed. Thus, the whole parser is correct. □

Some notes on the above proof.

- Some lemmas I apply have not been formally proven, but a lot of them have. Ones that I've proved include the concat inversion lemma, the bind combinator correctness lemma and the like.

- This proof proceeded using induction directly on the input `Val`. That's not how the serializer is written, although I have written versions of it where this is the case. In reality, the implementation is using the recursive parser / serializer. This is critical on the parsing side since it isn't clear that the recursive calls are structurally inductive. While it might be possible to flatten the parser out for a format this simple, that's something which will not scale so an alternative solution is required. The actual recursive combinator uses well-founded induction on the length of the encoding being parsed, and likewise the recursive serializer uses well-founded induction on the depth of the value being serialized. This is provided via a function argument, and intuitively, if we think of a `Val` as a binary tree with primitive values in the leaves, this is the height of the current tree. Both combinator implementations are "safe" in the sense that they explicitly check that the measure is decreasing before making a recursive call, causing an error otherwise. Even with the definition of `SerialVal` which doesn't use the recursive combinator, the proof would get stuck when having to deal with the recursive parse combinator.
- This gets even more tricky if the descriptor is required to serialize or parse the data. The naïve statement of the top-level correctness statement, something to the effect of `∀ (d : Desc), ParseOk (ParseVal d) (SerialVal d)` is ineffective since we don't have mechanisms to facilitate the change of descriptor during the recursive calls at the moment.

0.5 Recursive Combinator

Consider the current recursive combinator for parsing.

```
Program Fixpoint par_recur {R : Type} (underlying : Parser R -> Parser R)
(inp : Input) {measure (Length inp)} : ParseResult R :=
  underlying (fun rem => match decide ((Length rem) < (Length inp)) with
                      | left _ => par_recur underlying rem
                      | right _ => RecursiveProgressError "Parser.Recursive" inp
                                                          rem
                      end) inp.
Next Obligation.
  apply measure_wf.
  apply lt_wf.
Defined.

Definition ParseRecursive {R : Type} (underlying : Parser R -> Parser R) : Parser R :=
  fun inp => par_recur underlying inp.
```

Where `RecursiveProgressError` just checks if the measure didn't make progress or actually increased to tailor the error message.

This is extremely similar to the recursive serializer,

```
Program Fixpoint ser_recur {R : Type} {wfo : R -> Prop}
  (underlying : Serializer R wfo -> Serializer R wfo)
  (depth : R -> nat) (r : R) {measure (depth r)} : SerializeResult :=
  underlying (fun r__next => match decide ((depth r__next) < (depth r)) with
                          | left _ => ser_recur underlying depth r__next
                          | right _ => SerialRecursiveProgressError "Serializer.Recursive"
                                                                    depth r r__next
```

```
                        end) r.
Next Obligation.
  apply measure_wf.
  apply lt_wf.
Defined.

Definition SerialRecursive {R : Type} {wf : R -> Prop}
  (underlying : Serializer R wf -> Serializer R wf) depth : Serializer R wf :=
  ser_recur underlying depth.
```

In fact, I would argue that the recursive parser is structurally an instance of the serializer were the depth function is the length of the remaining input. After all, the length of a list can be though of as the depth of nested `Cons` constructors. The two definitions need to be separate only as a technical consideration to the fact that a `Serializer R wf` and `Parser R` aren't actually the same type.

I think that the correctness statement of these definitions isn't obvious, but there is clearly some link between them. What I've currently settled on (with some input for an LLM) is a statement that looks like this:

$$(\forall\ p\ s.\ \texttt{ParseOk}\ p\ s \to \texttt{ParseOk}\ (u_p\ p)\ (u_s\ s)) \to \\ \texttt{ParseOk}\ (\texttt{ParseRecursive}\ u_p)\ (\texttt{SerialRecursive}\ u_s\ \mathit{depth})$$

Recalling that the underlying parser and serializer have a hole in them, this hypothesis claims that if those underlying parser and serializer behave the same whenever the holes are filled with a matching parser / serializer pair then the behavior of them when wrapped in the recursive combinator will be correct.

While this isn't something that I have proved yet, I did write this out as an admitted theorem and am in the progress of using it in an attempt to prove the correctness of the simpler parser. While that proof isn't complete at the time of this writing, it is something that I believe I can prove. I've already solved the two base cases and am currently working on reasoning about the length-limiting behavior of a few other combinators as the last step.

I also don't know if the statement as written can be proved for these recursive definitions. Working with `Program Fixpoint` functions seems to be *difficult*, and I may need to write and prove a rewriting theorem first to contain the complexity of the underlying definitions automatically generated by Rocq. An alternative is to try and use an `Equations` version, which I have for the parser implementation and could easily write for the serializer. That should provide some better reasoning principles for these definitions.