Definition 0.1 (Encoding Field)**.** An encoding field is a tuple $(id, tag, value)$ which encodes the field number, tag and value of one field of a proto descriptor.

Just like how the message descriptor is a sequence of field descriptors, the encoding type of a message is an ordered list of encoding fields.

Definition 0.2 (Field Compatibility)**.** For fields $f_1 = (id_1, tag_1, v_1)$ and $f_2 = (id_2, tag_2, v_2)$, f_2 with descriptor d_2 is a compatible update to f_1 with descriptor d_1 if $id_1 = id_2$ and for all values v_1, $serialize_{d_1}\ f_1 = bs$ and there exists a v_2 such that $parse_{d_2}\ bs = v_2$ and $v_1 \prec v_2$ according to some relation $\prec$.

An example value relation is given in Section 1.1.

Definition 0.3 (Message Compatibility)**.** A protobuf message descriptor d_2 is a compatible update to d_1 if every field in d_1 has a compatible field in d_2.

Formally speaking, the message compatibility relation ($\preceq$) is given in Section 1.

Definition 0.4 (Tag Function)**.** The tag function tag : Type $\to$ Tag maps the `proto` level type to the corresponding tag as listed in Table **??**.

It is also important to understand the default values of each protobuf type. When a field is serialized with the default value or not set in the message, the parser places the default value in the resulting struct. The primary difference between an implicit field and an optional field is that the optional field can differentiate between the serialized field being omitted from the message or being present, but having the default value.

Definition 0.5 (Default Function)**.** The default function $default$: (t : proto type) $\to [\![t]\!]$ returns the default value for each type.

$$default(t) = \begin{cases} 0 & : \quad t \in \left\{ \begin{array}{l} \texttt{int32}, \texttt{int64}, \texttt{uint32}, \texttt{uint64}, \texttt{sint32}, \texttt{sint64}, \\ \texttt{fixed32}, \texttt{fixed64}, \texttt{sfixed32}, \texttt{sfixed64}, \texttt{enum} \end{array} \right\} \\ \texttt{false} & : \quad t \text{ is } \texttt{bool} \\ \text{“”} & : \quad t \text{ is } \texttt{string} \\ [\,] & : \quad t \text{ is } \texttt{bytes} \end{cases}$$

1 Protobuf Compatibility Relations

Compatibility between protobuf entities is defined with a series of relations. The first and most self-contained relation is the value relation, which relates two values at the specification level if serializing just a value (such as a variable width integer) and then parsing it to a different protobuf type would lead to a different value. After that is the protobuf type relation, which relates two protobuf type is any value of the first type is related to some value of the second type in the value relation. Since protobuf field and message definitions will not have concrete values associated with them, the type relation can be used in place of the value relation to make safe type changes. Finally, the message relation connects compatible messages such that all instances of the original message can be parsed into instances of the updated message.

1.1 Value Relation

The value relation is the base level of the compatibility relations, relating just two individual values. Protobuf values are modeled here as a specification level type in F* associated with a protobuf type as listed in Table 1. Finally, while protobuf decorators are technically part of the field level specification, they have been incorporated into the type information to allow the specification type into include `option` or `list` F* types.

Specification Type	Protobuf Type
$\mathbb{Z}$	`int32`, `uint32`, `sint32` `int64`, `uint64`, `sint64`
$\mathbb{B}$	`bool`
`string`	`string`
`list char`	`bytes`
–	`float`, `double`

Table 1: Description of the F* type corresponding to each protobuf type. Since F* doesn't support any type of floating point number, specification types cannot be provided for floating point protobuf types.

Definition 1.1 (Value Relation)**.** A protobuf value at the specification level v_1 with protobuf type τ_1 is related to another value v_2 at type τ_2, denoted $v_1 : \tau_1 \prec v_2 : \tau_2$ if $parse_{\tau_2}\ serialize_{\tau_1}\ v_1 = v_2$.

1.1.1 Basic Rules

The value relation is reflexive and transitive.

$$\textsc{Refl}\ \frac{}{v : \tau \prec v : \tau} \qquad \textsc{Trans}\ \frac{v_1 : \tau_1 \prec v_2 : \tau_2 \qquad v_2 : \tau_2 \prec v_3 : \tau_3}{v_1 : \tau_1 \prec v_3 : \tau_3}$$

1.1.2 String & Byte Rules

It is possible to convert been the `string` and `bytes` types, although the host program may have to deal with control character bytes being present in the resulting string.

$$\textsc{Str-Byte}\ \frac{}{v : \texttt{string} \prec v : \texttt{bytes}} \qquad \textsc{Byte-Str}\ \frac{}{v : \texttt{bytes} \prec v : \texttt{string}}$$

1.1.3 Integer Rules

As a shorthand, statements like `uint[n]` represent a variable length integer encoding into n bits. In order to result in valid protobuf types, $n \in \{32, 64\}$. The change width rules are designed to allow for both integer promotion and demotion while the rest of the rules express what happens when converting between integers using different encoding types for negative numbers.

With regard to the INT-CHG-W rule, I was originally concerned about negative 32 bit numbers being encoded into 5 bytes for a varint encoding and then becoming positive if parsed into 64 bit integer, but protobuf handles this by writing all negative `int[n]` into a full 10 bytes.

It is also worth noting that the expression $v : \tau$ actually refers to a value of type $[\![\tau]\!]$ with a label of the type. All of the integer types have $[\![\cdot]\!] = \mathbb{Z}$, as shown in Table 1, which allows for the free conversion between integer types. Finally, % refers to modulus using floored division, so $-1 \,\%\, 2 = 1$ and the rules are written with truncated integer division.

$$\textsc{Uint-Chg-W}\ \frac{v_2 = v_1 \,\%\, 2^m}{v_1 : \texttt{uint[n]} \prec v_2 : \texttt{uint[m]}} \qquad \textsc{Int-Chg-W}\ \frac{v_2 = (v_1 \,\%\, 2^{m-1} - 2^{m-1} \times \mathbb{1}[v_1 < 0]}{v_1 : \texttt{int[n]} \prec v_2 : \texttt{int[m]}}$$

$$\textsc{Sint-Chg-W}\ \frac{v_2 = v_1 \,\%\, 2^{m-1} - 2^{m-1} \times \mathbb{1}\left[\frac{v_1}{2^{m-1}} \,\%\, 2 = 1\right]}{v_1 : \texttt{sint[n]} \prec v_2 : \texttt{sint[m]}} \qquad \textsc{Uint-Int}\ \frac{v_2 = v_1 - 2^n \times \mathbb{1}[v_1 \geq 2^{n-1}]}{v_1 : \texttt{uint[n]} \prec v_2 : \texttt{int[n]}}$$

$$\textsc{Int-Uint}\ \frac{v_2 = v_1 + 2^n \times \mathbb{1}[v_1 < 0]}{v_1 : \texttt{int[n]} \prec v_2 : \texttt{uint[n]}} \qquad \textsc{Uint-Sint}\ \frac{v_2 = (-1)^{v_1} \times \left(\frac{v_1}{2}\right) - (v_1 \% 2)}{v_1 : \texttt{uint[n]} \prec v_2 : \texttt{sint[n]}} \qquad \textsc{Sint-Uint}\ \frac{v_2 = 2 \times |v_1| - \mathbb{1}[v_1 < 0]}{v_1 : \texttt{sint[n]} \prec v_2 : \texttt{uint[n]}}$$

$$\textsc{Int-Sint}\ \frac{v_2 = \begin{cases} (-1)^{v_1} \times \left(\frac{v_1}{2}\right) - (v_1 \,\%\, 2) & : \text{ if } v_1 \geq 0 \\ (-1)^{v_1} \times \left(v_1 + 2^{n-1} - \frac{v_1}{2}\right) & : \text{ otherwise} \end{cases}}{v_1 : \texttt{int[n]} \prec v_2 : \texttt{sint[n]}}$$

$$\textsc{Sint-Int}\ \frac{v_2 = \begin{cases} 2 \times |v_1| - \mathbb{1}[v_1 < 0] & : \text{ if } -2^{n-2} \leq v_1 < 2^{n-2} \\ 2 \times |v_1| - 2^n - \mathbb{1}[v_1 < 0] & : \text{ otherwise} \end{cases}}{v_1 : \texttt{sint[n]} \prec v_2 : \texttt{int[n]}}$$

1.1.4 Boolean Rules

Booleans can be converted to and from any of the integer types.

$$\textsc{Uint-Bool}\ \frac{v_2 = \begin{cases} \texttt{false} & : \text{ if } v_1 = 0 \\ \texttt{true} & : \text{ otherwise} \end{cases}}{v_1 : \texttt{uint[n]} \prec v_2 : \texttt{bool}} \qquad \textsc{Bool-Uint}\ \frac{v_2 = \mathbb{1}[v_1]}{v_1 : \texttt{bool} \prec v_2 : \texttt{uint[n]}} \qquad \textsc{Int-Bool}\ \frac{v_2 = \begin{cases} \texttt{false} & : \text{ if } v_1 = 0 \\ \texttt{true} & : \text{ otherwise} \end{cases}}{v_1 : \texttt{int[n]} \prec v_2 : \texttt{bool}}$$

$$\textsc{Bool-Int}\ \frac{v_2 = \mathbb{1}[v_1]}{v_1 : \texttt{bool} \prec v_2 : \texttt{int[n]}} \qquad \textsc{Sint-Bool}\ \frac{v_2 = \begin{cases} \texttt{false} & : \text{ if } v_1 = 0 \\ \texttt{true} & : \text{ otherwise} \end{cases}}{v_1 : \texttt{sint[n]} \prec v_2 : \texttt{bool}} \qquad \textsc{Bool-Sint}\ \frac{v_2 = -\mathbb{1}[v_1]}{v_1 : \texttt{bool} \prec v_2 : \texttt{sint[n]}}$$

1.1.5 Message & Enum Rules

Definitions for the structure of a message and enum are given in Section 1.3. The values are represented as a mapping from the id number of each field to the value and type of that field.

MSG

$$\frac{\forall\, i \in ids(v_{m_2}).\ (i \in ids(v_{m_1}) \wedge v_i \prec v_i') \vee v_i' = default(\tau_i')}{v_{m_1} = \{id_1 : (v_1, \tau_1) \ldots, id_n : (v_n, \tau_n)\} : \texttt{MSG}\ m_1 \prec v_{m_2} = \left\{id_1' : (v_1', \tau_1') \ldots, id_m' : (v_m', \tau_m')\right\} : \texttt{MSG}\ m_2}$$

ENUM

$$\frac{v \in vals(e_2)}{v : \texttt{ENUM}\ s_1\ e_1 \prec v : \texttt{ENUM}\ s_2\ e_2}$$

1.1.6 Decorator Rules

Since the specification type can include `option`'s and `list`'s, rules for handling these must also be present in the value relation. These rules are grouped into introduction rules, which take implicit values and make them optional or repeated, pass through rules which enable the above rules to operate on these modified types.

There really should be a rule which can introduce a `None`, but since implicit fields are modeled as just the spec type, this is difficult. I will likely need to model both implicit and optional fields with an `option`, add an annotation to the protobuf type and provide a function which introduces a default value for each type.

REP-PASS

$$\frac{v_1 : \tau_1 \prec v_1' : \tau_2 \quad \cdots \quad v_n : \tau_1 \prec v_n' : \tau_2}{[v_1; \ldots; v_n] : \texttt{REP}\ \tau_1 \prec [v_1'; \ldots; v_n'] : \texttt{REP}\ \tau_2}$$

OPT-SOME-PASS

$$\frac{v_1 : \tau_1 \prec v_2 : \tau_2}{(\texttt{Some}\ v_1) : \texttt{OPT}\ \tau_1 \prec (\texttt{Some}\ v_2) : \texttt{OPT}\ \tau_2}$$

OPT-NONE-PASS

$$\frac{}{\texttt{NONE} : \texttt{OPT}\ \tau_1 \prec \texttt{NONE} : \texttt{OPT}\ \tau_2}$$

OPT-INTRO

$$\frac{}{v : \tau \prec (\texttt{Some}\ v) : \texttt{OPT}\ \tau}$$

REP-INTRO

$$\frac{}{v : \tau \prec [v] : \texttt{REP}\ \tau}$$

MISSING-IMP

$$\frac{}{\texttt{NONE} : \tau_1 \prec default(\tau_2) : \tau_2}$$

1.2 Type Relation

This relation relates types which can be converted via the value relation.

Definition 1.2 (Type Relation)**.** The type relation relates two protobuf type τ_1 and τ_2, denoted $\tau_1 \propto \tau_2$ if for all $v_1 : \tau_1$ there exists a $v_2 : \tau_2$ such that $v_1 : \tau_1 \prec v_2 : \tau_2$.

1.2.1 Base Type Rules

$$\textsc{Str-Byt-T}\ \frac{}{\texttt{string} \propto \texttt{bytes}} \qquad \textsc{Byt-Str-T}\ \frac{}{\texttt{bytes} \propto \texttt{string}} \qquad \textsc{Int-Int-T}\ \frac{\tau_1, \tau_2 \in \left\{ \begin{array}{c} \texttt{int32}, \texttt{int64}, \texttt{uint32}, \texttt{uint64} \\ \texttt{sint32}, \texttt{sint64}, \texttt{bool} \end{array} \right\}}{\tau_1 \propto \tau_2}$$

1.2.2 Message & Enum Rules

$$\textsc{Msg-T}\ \frac{m_1 \preceq m_2}{\texttt{MSG}\ m_1 \propto \texttt{MSG}\ m_2} \qquad \textsc{Enum-T}\ \frac{vals(e_1) \subset vals(e_2)}{\texttt{ENUM}\ s_1\ e_1 \propto \texttt{ENUM}\ s_2\ e_2}$$

1.2.3 Decorator Rules

Since the type relation is designed for checking for safe type changes, it also need to ensure that decorator restrictions like moving from a repeated field to a singleton one aren't violated.

$$\textsc{Opt-Add-T}\ \frac{}{\tau \propto \texttt{OPT}\ \tau} \qquad \textsc{Opt-Rm-T}\ \frac{}{\texttt{OPT}\ \tau \propto \tau} \qquad \textsc{Opt-T}\ \frac{\tau_1 \propto \tau_2}{\texttt{OPT}\ \tau_1 \propto \texttt{OPT}\ \tau_2}$$

$$\textsc{Rep-Add-T}\ \frac{}{\tau \propto \texttt{REP}\ \tau} \qquad \textsc{Rep-T}\ \frac{\tau_1 \propto \tau_2}{\texttt{REP}\ \tau_1 \propto \texttt{REP}\ \tau_2}$$

1.3 Message Relation

The descriptor compatibility relation $\preceq$ is defined as below, but first it is important to discuss the structure of fields and messages.

Each field is one of three different type of fields, the standard scalar field, a map field or a `oneof` field. Specifically, the inductive type for each style of field contains these pieces of information:

- `FIELD`: Contains the name of the field, the identifying field number and protobuf type (containing the decorator as well, possibly `OPT` or `REP`).
- `MAP`: Contains the name of the field, the identifying field number and two protobuf types, one for the keys and one of the values.
- `ONEOF`: Contains a name and a set of other fields.

A message descriptor and an enum descriptor are modeled one step above a field, containing the following pieces of information:

- `MSG`: Contains the name of the message, a set of reserved indices and then a set of field definitions.
- `ENUM`: Contains a set of name, id number pairs.

1.3.1 Basic Rules

A compatibility relation is both reflexive and transitive, although it is neither symmetric nor antisymmetric.

$$\textsc{Refl-M}\ \frac{}{\texttt{MSG}\ s\ r\ f \preceq \texttt{MSG}\ s\ r\ f} \qquad \textsc{Refl-E}\ \frac{}{\texttt{ENUM}\ s\ v \preceq \texttt{ENUM}\ s\ v}$$

$$\textsc{Trans-M}\ \frac{\texttt{MSG}\ s_1\ r_1\ f_1 \preceq \texttt{MSG}\ s_2\ r_2\ f_2 \qquad \texttt{MSG}\ s_2\ r_2\ f_2 \preceq \texttt{MSG}\ s_3\ r_3\ f_3}{\texttt{MSG}\ s_1\ r_1\ f_1 \preceq \texttt{MSG}\ s_3\ r_3\ f_3}$$

$$\textsc{Trans-E}\ \frac{\texttt{ENUM}\ s_1\ v_1 \preceq \texttt{ENUM}\ s_2\ v_2 \qquad \texttt{ENUM}\ s_2\ v_2 \preceq \texttt{ENUM}\ s_3\ v_3}{\texttt{ENUM}\ s_1\ v_1 \preceq \texttt{ENUM}\ s_3\ v_3}$$

1.3.2 Field Update Rules

For the moment, names are recorded the protobuf field descriptors for accuracy and to allow the relations to expand in the future. However, since names are recorded in the encoded message, they are currently allowed to freely change.

$$\textsc{Field-Name}\ \frac{}{\texttt{MSG}\ s_m\ r\ (f \cup \{\texttt{FIELD}\ s_f\ id\ \tau\}) \preceq \texttt{MSG}\ s_m\ r\ (f \cup \{\texttt{FIELD}\ s'_f\ id\ \tau\})}$$

$$\textsc{Map-Name}\ \frac{}{\texttt{MSG}\ s_m\ r\ (f \cup \{\texttt{MAP}\ s_f\ id\ \tau_1\ \tau_2\}) \preceq \texttt{MSG}\ s_m\ r\ (f \cup \{\texttt{MAP}\ s'_f\ id\ \tau_1\ \tau_2\})}$$

$$\textsc{Oneof-Name}\ \frac{}{\texttt{MSG}\ s_m\ r\ (f \cup \{\texttt{ONEOF}\ s_f\ f_o\}) \preceq \texttt{MSG}\ s_m\ r\ (f \cup \{\texttt{ONEOF}\ s'_f\ f_o\})}$$

$$\textsc{Field-Type}\ \frac{\tau_1 \propto \tau_2}{\texttt{MSG}\ s_m\ r\ (f \cup \{\texttt{FIELD}\ s_f\ id\ \tau_1\}) \preceq \texttt{MSG}\ s_m\ r\ (f \cup \{\texttt{FIELD}\ s_f\ id\ \tau_2\})}$$

$$\textsc{Map-Type-F}\ \frac{\tau_1 \propto \tau'_1 \qquad \tau_2 \propto \tau'_2 \qquad \tau_1 \neq \texttt{bytes}}{\texttt{MSG}\ s_m\ r\ (f \cup \{\texttt{MAP}\ s_f\ id\ \tau_1\ \texttt{IMP}\ \tau_2\}) \preceq \texttt{MSG}\ s_m\ r\ (f \cup \{\texttt{MAP}\ s_f\ id\ \tau'_1\ \texttt{IMP}\ \tau'_2\})}$$

1.3.3 Oneof Rules

The `oneof` rules are a bit complicated because a `oneof` struggles against the notion that the field relations only connects one field to one field by encapsulating several fields into a singular field.

ONEOF-INTRO-FIELD

$$\frac{m \in \{\texttt{IMP}, \texttt{OPT}\}}{\texttt{MSG}\ s_m\ r\ (f \cup \{\texttt{FIELD}\ s_f\ id\ (m\ \tau)\}) \preceq \texttt{MSG}\ s_m\ r\ (f \cup \{\texttt{ONEOF}\ s_o\ \{\texttt{FIELD}\ s_f\ id\ (m\ \tau)\}\})}$$

ONEOF-ADD-F

$$\frac{m \in \{\texttt{IMP}, \texttt{OPT}\} \qquad s_n \notin names(f \cup f_o) \qquad id_n \notin ids(f \cup f_o)}{\texttt{MSG}\ s_m\ r\ f \cup \{\texttt{ONEOF}\ s_f\ f_o\} \preceq \texttt{MSG}\ s_m\ r\ f \cup \{\texttt{ONEOF}\ s_f\ (f_o \cup \{\texttt{FIELD}\ s_n\ id_n\ (m\ \tau)\})\}}$$

ONEOF-INTRO-NEW

$$\frac{\begin{array}{c} s_f \notin names(f) \qquad ids(f) \cap ids(f_o) = \varnothing \qquad names(f) \cap names(f_o) = \varnothing \\ \texttt{REP} \notin dec(f_o) \qquad \forall f_n \in f_o.\ f \neq \texttt{MAP} \end{array}}{\texttt{MSG}\ s_m\ r\ f \preceq \texttt{MSG}\ s_m\ r\ (f \cup \{\texttt{ONEOF}\ s_f\ f_o\})}$$

ONEOF-ELIM

$$\frac{}{\texttt{MSG}\ s_m\ r\ (f \cup \{\texttt{ONEOF}\ s_f\ f_o\}) \preceq \texttt{MSG}\ s_m\ r\ (f \cup \{f_o\})}$$

ONEOF-FEILD-UPDATE

$$\frac{\texttt{MSG}\ \text{“”}\ \varnothing\ \{f\} \preceq \texttt{MSG}\ \text{“”}\ \varnothing\ \{f'\}}{\texttt{MSG}\ s\ r\ (f_m \cup \{\texttt{ONEOF}\ s_f\ (f_o \cup \{f\})\}) \preceq \texttt{MSG}\ s\ r\ (f_m \cup \{\texttt{ONEOF}\ s_f\ (f_o \cup \{f'\})\})}$$

1.3.4 Field & Enum Rules

Messages are similar to records in F* or other functional languages and message compatibility can take the same two routes that record subtyping can: width compatibility and depth compatibility. In width compatibility updates, new fields are added to the message, making the updated version a strict superset of the original message. In depth compatibility updates, the number of fields remains the same, but other attributes of the field are updated compatible ways.

MSG-WIDTH-FIELD

$$\frac{s_f \notin names(f) \qquad id \notin ids(f)}{\texttt{MSG}\ s_m\ r\ f \preceq \texttt{MSG}\ s_m\ r\ (f \cup \{\texttt{FIELD}\ s_f\ id\ \tau\})}$$

MSG-WIDTH-DEPTH

$$\frac{\texttt{MSG}\ \text{“”}\ \varnothing\ \{f\} \preceq \texttt{MSG}\ \text{“”}\ \varnothing\ \{f'\}}{\texttt{MSG}\ s_m\ r\ (fs \cup \{f\}) \preceq \texttt{MSG}\ s_m\ r\ (fs \cup \{f'\})}$$

The enum rules are similar to the message rules at least in the sense that enums can be updated to have new fields without issue. However, when considering removing fields from an enum definition, it is important to know if the enum is treated as open or closed [**EnumBehavior**]. By default, `proto3` treats all enums as open, so it should be possible to write a rule allowing for the deletion of an enum field. This naturally aligns with how `go` handles enums, since it doesn't support closed enums. Since protobuf enums don't contain extra information, no depth rules is allowed.

ENUM-WIDTH

$$\frac{vals(e_1) \subset vals(e_2)}{\texttt{ENUM}\ s_1\ e_1 \preceq \texttt{ENUM}\ s_2\ e_2}$$

1.3.5 Reserved Field Rules

$$\frac{r \subset r'}{\texttt{MSG}\ s\ r\ f \preceq \texttt{MSG}\ s\ r'\ f}\ \textsc{Reserved-Add} \qquad \frac{r \supset r'}{\texttt{MSG}\ s\ r\ f \preceq \texttt{MSG}\ s\ r'\ f}\ \textsc{Reserved-Rm}$$

1.3.6 Valid Descriptors

Definition 1.3 (Valid Protobuf Descriptors)**.** A protobuf message descriptor is *valid*, denoted $v(d)$ if $\texttt{MSG}\ ``"\ \varnothing\ \varnothing \preceq \texttt{MSG}\ s\ r\ f$.

1.3.7 Compatibility Theorem

The main field-level compatibility theorem is stated below.

Theorem 1 (Field Compatibility)**.** *If $d_1 \preceq d_2$ then for all binary strings bs such that $\texttt{parse}_{d_1}\ bs = \texttt{Some}\ v_1$, $\texttt{parse}_{d_2}\ bs = \texttt{Some}\ v_2$ and $v_1 \prec v_2$*