Cap'n Proto Elm

Author: Prasanth Somasundar <mezuzza@live.com>

Status: Implementing

Background

Cap'n Proto (capnp) is a binary data interchange format that like ProtoBuf provides a number of advantages over traditional text based formats. These include speed, type safety, and easier inter-process communication.

Elm is a language designed to compile to JavaScript and run on the browser. Historically, JSON has been the interchange format used in Elm. However, this monopoly is being challenged.

For my own purposes, I want to explore what it feels like to use a stack with Cap'n Proto. Because I don't like doing things easily and for a bit of fun, I've decided to build out the Cap'n Proto language plugin for Elm.

Objective

Build a plugin for the Cap'n Proto compiler that supports the Elm programming language.

Design

This document specifies the design the following:

- A prototype for the Elm Cap'n Proto API

The following are non-goals for this design:

- Cap'n Proto RPC – This will be handled in a future document. See the design section for how we will handle interfaces.
- Performance – This solution uses `Array (Array Int)` to hold data. This is clearly not good for performance and will only be used for prototyping the api.

Priorities for this project

1. The API should provide at least one way to handle any feature.
2. The API should handle a "good" schema ergonomically.
3. The API should be idiotproof. There shouldn't be an easy way to shoot yourself in the foot.

4. The API should be easy to read - even if the types are somewhat complex.

Capnp Compiler Plugin Design

A capnp compiler plugin simply reads a capnp from stdin. The schema for this capnp is specified in the capnp repository. This plugin will be built in C++. See appendix for why this decision was made.

Formatting

In order to allow the compiled code to be human readable and not worry about formatting in the plugin layer, we'll run the generated code through elm-format before creating a file. Additionally, we'll guard this with a flag `--format-code`.

Elm annotations

We will produce two annotations that are consumed by the compiler plugin:

```
annotation module(file): Text;
annotation overrideName(*): Text;
```

The elm package should be configurable by the capnp schema author. To that end, we'll accept a file annotation in the schema file that specifies the module prefix explicitly. This annotation will be used like so:

```
# bar.capnp
using Elm = import "capnp/elm.capnp";

$Elm.module("Foo.Bar");
```

As for the `override_name` annotation, we will provide it as an escape hatch for users to avoid name collisions in their produced code.

Produced Files

The compiler plugin produces a directory for each schema file. Inside this directory, each file generates a unique module that's usable by Elm. The example above will create the module `Foo.Bar`.

Elm Capnp API

Before getting into the actual API, let's define an example struct to use as motivation for Capnp features:

```
# Example borrowed from Capnp Haskell implementation with modification
using Elm = import "capnp/elm.capnp";
$Elm.module("AddressBook");

struct Person {
  id @0 :UInt32;
  name @1 :Text;
  email @2 :Text;
  mainPhone @8 :PhoneNumber;
```

```
  phones @3 :List(PhoneNumber);
  picture @7 :Data;

  struct PhoneNumber {
    number @0 :Text;
    type @1 :Type;

    enum Type
     $Elm.overrideName("PhoneType") {
      mobile @0;
      home @1;
      work @2;
    }
  }

  employment :union {
    unemployed @4 :Void;
    employed :group {
      employer @5 :Text;
      employeeId @6 :UInt32;
    }
  }
}

struct AddressBook {
  people @0 :List(Person);
}
```

General Naming

In general, the names in a module cannot conflict. As `_` is an illegal character in Cap'n Proto names, we'll use it as a safe separator in names. To avoid very long and complex names, we will be flattening out all references in the schema.

For example, the name for the struct in the capnp schema is AddressBook.Person.PhoneNumber while the Elm type will be AddressBook.PhoneNumber. This can obviously cause name collisions, but you can use `$Elm.overrideName` in the schema to correct these issues.

Primitive Type Mapping

The type mapping below holds all the type mappings from Capnp's primitive types to Elm's types. They are pretty straightforward.

Capnp Type	Elm Type
Void	Unit*

Bool	Bool
IntXX	Capnp.IntXX
UIntXX	Capnp.UIntXX
FloatXX	Float
Text	String
Data	Bytes

*Void generally will not be encoded in Capnp by this library. Void can be thought of as Unit in Elm. Since this type carries no information, fields for this type will not have a generated code that can access the field. If you wish to access the field, just use (). It saves you time.

Unfortunately, there does not seem to exist a great way to produce sized numerical values in Elm. The typical way to handle this seems to be using the included `Int` type. This doesn't provide type guarantees, but it does keep the APIs simpler. However, since the APIs provided by this library are fairly complex and selecting the size of number representations matters when you are talking about binary data, I think it would be best to avoid providing easy ways to shoot yourself in the foot. For this reason, a fixed width numeric library will be built alongside this one.

Of special note, since Javascript and Elm by extension uses 64 bit floats to represent all numbers, special care will be used in handling 64-bit integers.

Enums

An enumeration in Capnp will be converted to a type in Elm as follows:

```
module AddressBook exposing(..)

-- The original name was `Type`. Overridden by the annotation.
-- No matter what, the type's name is capitalized and the functions are
-- lowercased.
type PhoneType
  = Mobile
  | Home
  | Work

-- Obvious constructions to and from Ints. Use is not suggested,
-- but provided for convenience. I'm tempted to remove them altogether,
-- but I'm almost certain they will be written a million times.
phoneType_toInt : Type -> Int
phoneType_fromInt : Int -> Maybe Type
```

```
-- Full list used to provide easy ranging through the list.
phoneType_asList : List Type

-- Obvious conversations to and from string.
phoneType_toString : Type -> String
phoneType_fromString : String -> Maybe Type
```

As you can see we create a set of utility functions that help clients use the data type as an enumeration. Currently, that list is limited to:

- fromInt/toInt -> converts the enum to and from an integer. Use of these functions is discouraged from Capnp's side, but should be available for when use is necessary.
- toString/fromString -> converts to and from a string.
- asList -> allows you to create an ordered list of the enum. You can use List.take and List.drop to generate ranges as desired.

Lists

Lists act almost like a special message. We create a module `Capnp.List` that is used by all messages. Inside the module, we expose an opaque type and the following functions:

```
module Capnp.List exposing(List, ...)

import List

-- Data definition
type List a = ...

-- Virtual list used in virtual proto. See Set/Modify implementation
-- details.
type alias Virtual_ a = List (Maybe a)

-- Easy way to create a simple list.
singleton : a -> List a

-- Obvious conversions to and from a native Elm list.
fromList : List.List a -> List a
toList : List a -> List.List a

-- Query/Consumption
-- Note that we are clearly missing a `map` function. This is
-- intentional. While inconvenient, `map` doesn't make sense in this
-- context as we are manipulating data in a fixed width array that
-- represents a location in memory.
--
-- We'll handle this question more closely in the setters sections
```

```
get : Int -> List a -> a
foldl : (a -> b -> b) -> b -> List a -> b
foldr : (a -> b -> b) -> b -> List a -> b
length : List a -> Int
all : (a -> Bool) -> List a -> Bool
any : (a -> Bool) -> List a -> Bool
isEmpty : List a -> Bool
```

Unions

Unions are declared as a standard data type in Elm. The union's name will be used to generate the data type. One cannot access the fields declared inside a union without first unpacking the union itself. The union's data constructors will be prefixed with the union's name to avoid conflicts.

As a small optimization, a field with type `Void` will not have any payload.

```
type Employment
  = Employment_Unemployed
  | Employment_Employed (Field Person Employed)
```

Unnamed Unions

Since every struct can have an unnamed union, we'll need to choose a name. We'll use `Which_` due to [this conversation].

Groups

Since groups are bound to the surrounding struct, we'll produce a new type alias based on the name of the group inside the file of the parent struct. Thankfully, since groups are also type aliases, they do not produce Constructors that could clash with the surrounding union.

```
type alias Employed =
  { employer : String
  , employeeId : Capnp.UInt32
  }
```

Constants

Not much to say. These will be produced in their parent modules as constants.

Structs

Structs will be backed by a byte buffer. For our prototype, we'll use `Array Bytes` to handle each segment. Each struct produces a single file. Most of the useful code is produced by a library file in pure Elm. This file defines how we interact with structs in Elm code.

Elm Library Code

```
module Capnp.Struct exposing(..)

import Bytes
```

```
import Bytes.Decode as Decode
import Bytes.Encode as Encode

type alias Data = Array Bytes.Bytes

-- Represents structs of type s. Should be treated as opaque by both generated
and client code
type Struct s = Struct
  { -- Represents the raw data that defines the struct.
    -- This doesn't change per message
    data : Data
  , --  Definition of the struct as a type aliased record.
    fields : s
  , -- Index into `data` from which to begin reading.
    viewOffset : (Int, Int)
  , -- Amount of data traversed across this message to arrive at
    -- this struct.
    currentTraversalDistance : Int
  , -- Maximum amount of data in bytes to traverse before killing the
    -- traversal.
    traversalLimit : Int
  }

-- Represents a method call that allows us to access a field in a struct
-- of type s. The resulting call produces a type t. Used to traverse
-- `Struct.data`.
-- v and t' are used to traverse the virtual proto. See set/modify
-- implementation details.
type FieldData s v t t' = FieldData
  { segment : Int
  , offset : Int
  , decoder : Decode.Decoder t
  , encoder : Encode.Encoder t
  , virtual : (v -> Maybe t')
  }

-- Value fields
type alias Field s v t = FieldData s v t t

type FieldAccessError
  = InvalidPointer
  | TraversalLimitExceeded

-- Pointer field type aliases
```

```
-- s and v are the original struct and the virtual struct.
-- s' and v' are the retrieved struct and the retrieved struct's virtual
-- implementation.
type alias StructField s v s' v' =
  FieldData s v (Result FieldAccessError (Capnp.Struct s')) v'
type alias ListField s v s' v' =
  FieldData s v (Result FieldAccessError (Capnp.List.List a)) (Capnp.List.List
(Maybe v'))
type alias TextField = Result FieldAccessError String
type alias BytesField = Result FieldAccessError Bytes.Bytes

-- Initializes a new message with root struct s.
type alias InitOptions s =
  { struct : s
  , segmentSize : Int
  , numSegments : Int
  }
init : InitOptions s -> Struct s

-- See below
get : (s -> Field s a) -> Struct s -> a

-- Setters below.
-- All value types have similar setters and have therefore been omitted.
setInt8 : (s -> Field s Int8)
       -> Int8
       -> Struct s
       -> Struct s

-- Pointer types can fail field access and thus must return a result.
setList : (s -> ListField a)
       -> Capnp.List.List a
       -> Struct s
       -> Result FieldAccessError (Struct s)
setBytes : (s -> BytesField)
        -> Bytes
        -> Struct s
        -> Result FieldAccessError (Struct s)
setText : (s -> TextField)
        -> String
        -> Struct s
        -> Result FieldAccessError (Struct s)
setStruct : (s -> StructField s')
        -> s'
```

```
        -> Struct s
        -> Result FieldAccessError (Struct s)

-- Modifiers added to avoid double traversal. They will be 1:1 with setters.
-- Some have been omitted to avoid repetition:
modifyInt8 : (s -> Field s Int8)
          -> (Int8 -> Int8)
          -> Struct s
          -> Struct s

-- Pointer field modification can fail. Thus, the modification should be
-- allowed to fail.
modifyStruct : (s -> StructField s')
            -> (s' -> Result FieldAccessError s')
            -> Struct s
            -> Result FieldAccessError (Struct s)
```

Type of first argument to get and set

The type of get and set have been specifically chosen to work well with Elm's record syntax. You can use `.field` to get the `Field` for a given message. As an example:

```
getFirstPhoneNumber : Struct Person -> String
getFirstPhoneNumber =
  AddressBook.person_get .phones
  >> Struct.List.get 0
  >> AddressBook.personNumber_get .number
  >> Result.default "000-000-0000"
```

The setters in particular are unfortunate as they require some specialized code and there is no scheme to do this in Elm. All pointer field setters also produce `Result` values as pointers are validated on access and an invalid pointer would cause a runtime failure that we need to handle properly.

Implementation of Get

The implementation of Get will rely on the `Bytes.Decode.bytes` decoder. This decoder does seem to be fairly cheap as it only constructs a new dataview. This is essentially an index into the underlying byte buffer in constant time.

Implementation of Set/Modify

Set is a much more complicated problem. There are three possible solutions to updating values as I see them:

1. Do not allow inplace updates at all.
2. Allow asynchronous inplace updates through `Cmd`s.
3. Create a "virtual proto" on top of the raw bytes that represents modifications to the proto.

The first two are possible solutions, but the beauty of the final one is that there is no dependence on mutability at all. We can keep immutable data structures that feel like they operate in Elm. When transmitting

the data over the wire, we can always collapse the virtual proto into the byte array. When reading, there will be a constant overhead cost that should be negligible.

The major issue is complexity. This makes the types in the framework way harder to understand from simple reading. There will be considerable effort required to communicate their meanings in type errors and documentation.

This virtual proto will be represented by a `Maybe` with a special type that mirrors the original proto.

Generated Code

This will produce an Elm record for each struct in the original code.

```
module AddressBook exposing (..)

type Person_Virtual_ = Person_Virtual_
  { id : Maybe Capnp.UInt32.UInt32
  , name : Maybe String
  , email : Maybe String
  , phones : Maybe (Capnp.List.Virtual_ PhoneNumber_Virtual_)
  , pictures : Maybe Bytes.Bytes
  , employment : Maybe Employment_Virtual_
  }

type alias Person_ =
  { id : Capnp.Field Person Capnp.UInt32.UInt32
  , name : Capnp.Field Person TextField
  , email : Capnp.Field Person TextField
  , phones : Capnp.Field Person (ListField PhoneNumber)
  , pictures : Capnp.Field Person BytesField
  , employment : Capnp.Field Person (StructField Employment)
  }

type Person = Person Person_

-- Getters and setters listed in Capnp.Struct module repeated here.
-- They will have to be wrapped due to the opaqueness of `Person`.
-- One example given, the remainder have been omitted.
person_get : (Person_ -> Capnp.Field Person a) -> Struct Person -> a

decoder : Capnp.Decoder Person
encoder : Capnp.Encoder Person
```

There are a couple of oddities with this code. Specifically, the `Maybe` produced by each field accessor that is encoded with a pointer. Capnp lazily validates pointers on access and thus cannot guarantee that an access succeeds.

Note that the employment field actually produces a function. This is to provide the simplest way to define a struct's fields without requiring new compiler features or complicating the type of `Field`.

Interfaces

If we encounter an interface, we will produce a warning. However, that is the extent to which we will handle them in the current document. An interface is fairly useless without the RPC protocol.

AnyPointer

An AnyPointer can have the type of a struct, interface, list, or blob. As such, we'll model it directly with bytes and convert to and fro using functions.

```
module Capnp.AnyPointer exposing (..)
import Bytes

type AnyPointer = AnyPointer Bytes.Bytes

-- conversions
fromList : Capnp.List -> AnyPointer
asList : AnyPointer -> Capnp.List

fromStruct : Capnp.Struct m -> AnyPointer
asStruct : m -> AnyPointer -> Capnp.Struct m

fromText : String -> AnyPointer
asText : AnyPointer -> String

fromBytes : Bytes.Bytes -> AnyPointer
asBytes : AnyPointer -> Bytes.Bytes
```

Messages

A message can be encoded or decoded using the provided encoders and decoders generated alongside the root struct. These encoders and decoders will be consumed by the packages `Capnp.Encode` and `Capnp.Decode`. These provide two functions each – one for packed and one for unpacked.

```
module Capnp.Encode exposing (..)

type Encoder = ...

encode : Encoder s -> Capnp.Struct s -> Bytes
encodePacked : Encoder s -> Capnp.Struct s -> Bytes
```

```
module Capnp.Decode exposing(..)

-- The decoders are slightly different as they need some configuration to
```

```
-- allocate the buffers for the data.

type Decoder = ...

decode : Capnp.Struct.InitOptions -> Decoder s -> Bytes -> Capnp.Struct s
decodePacked : Capnp.Struct.InitOptions
             -> Decoder s
             -> Bytes
             -> Capnp.Struct s
```

Motivating Examples

Read

Security

There are three security considerations listed in the encoding spec. We should tackle these and any others here.

Pointer Invalidation

This is handled whenever we dereference a string, byte, list, or struct fields. In all cases, field accessors will produce a type `Result FieldAccessError a`. If the pointer is invalid, we'll produce an `Err InvalidPointer`.

Amplification Attack

This is tracked on a per message basis. This means that calls to get on sub-structs have the quota counted against them. Either way, at 64 Mib, I doubt that this is a problem for most cases. Additionally, we'll allow configuration of this in `InitOptions`.

Stack overflow DoS Attack

We'll avoid this largely by creating self referential tail recursive functions. The Elm compiler handles tail call elimination for us in these cases. Assuming that we are always jumping to new pointers and those pointers are valid, we will hit the Amplification limit way before the Stack overflow.

Appendix

Why C++ for language for compiler plugin?

This question has 3 potential resolutions:

1. Self-hosting – This is a pain even normally, but generally can be overcome. Elm however is not a general-purpose language; It's designed for the web. This means that it doesn't have great ways to handle stdin and stdout.
2. Use a non-c++ language (probably read Haskell) – These apis aren't necessarily stable nor are they consistently supported. Features may be missing.

   As for the Haskell implementation in particular, having spoken to Ian Denhardt, it seems that it has everything that I would need for the implementation. It is an option.

3. Use c++ – While not a functional language or self-hosting one, seems like this doesn't have any show stoppers like the ones listed above.

Why not use two annotations like java annotation in ProtoBuf?

No idea. This seems sufficient and simpler. Somewhat related to below.

Why not use a single file for each struct?

There are a few interesting pros and cons when producing a single file per struct:

Pros:

1. Simpler namespacing which matches Elm user intuitions
2. Smaller and more intuitive names

Cons:

1. Elm doesn't allow mutually recursive modules - for good reason. However, two capnp structs can clearly be mutually recursive - also for good reason. This means that if we want a single import per struct, we're forced to arbitrarily genericize one or all of the recursive values.
   This *can* be done, but it is definitely more complexity to manage.
2. Giant list of import statements to use even basic functionality. This would likely require a few aliases in the top level module that looks like a single file per capnp schema file anyway.