ASN.1 Encoding Schemes Done Right Using CMPCT

Mark Tullsen

Galois, Inc.

Bx 2019
Outline

1. Background
2. The Big Idea: Generalizing Encoders and Decoders
3. The CMPCT DSL (Bijections All The Way Down)
4. Structuring 'Bad': the "Extend with Errors" type
5. Expressiveness & Use Cases
6. Conclusion
1 Background

2 The Big Idea: Generalizing Encoders and Decoders

3 The CMPCT DSL (Bijections All The Way Down)

4 Structuring 'Bad': the "Extend with Errors" type

5 Expressiveness & Use Cases

6 Conclusion
**J2735 Standard** specifies the "Basic Safety Message" used in Vehicle to Vehicle (V2V) communications.

- Specified using ASN.1
- Uses the UPER encoding rule set

Creating, proving-correct, testing inter-operatibility of J2735 encoder/decoder implementations
The Immediate Need

- We want to generate *invalid* bit-streams for J2735
  - want "good" (?!?) coverage
  - know we’ve covered every class of error?
- Nice to have a solution that works
  - for any ASN.1 type (not ad hoc for J2735)
  - for other "encoding rule" schemes besides UPER
The Original Approach

\[
\text{invalid } bs = \text{decode}(bs) == \text{NoDecodeValue}
\]

\[
\text{testVectors} = \text{filter} \quad \text{invalid} \\
\quad \quad \text{(generateBitStreamsUpToLen 600)}
\]

... is not ideal:

- wrong if decode implementation is wrong!
- cannot distinguish the kind or severity of errors
- inefficient!
- will be primarily "extraneous bits at end" errors
  - error at 10th bit in the bit-stream of 600 bits!
Possibilities?

- Remove dependence on decode implementation
- Filter out "similar" bitstreams
  - we decode in byte order, 2nd errors are rarely important.
  - if equivalent up to the first error, different suffixes don’t matter!
- For each "logical error"
  - generate no more than $n$ invalid vectors
  - generate at least $n$ invalid vectors
ASN.1 (Abstract Syntax Notation One)

- A data description language
  - used to define hundreds of protocols and data-formats
- Expressive: a powerful set of types and constraints for describing data.
- Abstract: not tied to a single language or concrete representation
- Versatile: a variety (!) of encoding methods, specified separately from the definition of data types: (BER, DER, XER, PER, UPER, OER, ...)
- Overly complicated
  - Has been evolving for decades!
  - Two compilers, at most, support the full ASN.1 language
1 Background

2 The Big Idea: Generalizing Encoders and Decoders

3 The CMPCT DSL (Bijections All The Way Down)

4 Structuring 'Bad': the "Extend with Errors" type

5 Expressiveness & Use Cases

6 Conclusion
Current Approach

Values

Good:

{Bad}

Bitstreams

enc

dec
Make \( \text{dec} \) injective.
Generalizing Encode

Values

Good:

Bitstreams

valid

Bad:

encG

dec

encB

This Talk in a Nutshell

\[
\begin{align*}
\text{Values} & \quad \text{Bitstreams} \\
\text{Good:} & \quad \text{Valid} \\
\text{Bad:} & \quad \text{Invalid}
\end{align*}
\]
if we

- add structure to Bad (i.e., static types)
- can define $\text{encG}$, $\text{encB}$, $\text{dec}$
  - simultaneously (ensuring consistency of $\text{enc}$ and $\text{dec}$)
  - compositionally (i.e., build larger $\text{enc/dec}$ pairs from smaller)
then we

- can generate exactly what we want in Bad using Quickcheck.
- use encB to generate our invalid test vectors
- are not dependent upon an implementation of dec
- can extract the original enc/dec routines from the generalized ones
Background

The Big Idea: Generalizing Encoders and Decoders

The CMPCT DSL (Bijections All The Way Down)

Structuring 'Bad': the "Extend with Errors" type

Expressiveness & Use Cases

Conclusion
CMPCT: a domain specific language (DSL) for defining "compact" representations of abstract values.

- Defines correct, safe encoding/decoding pairs by design.
- It has morphed into a language of bijections.

"CMPCT" = compact("compact") -- not an acronym
CMPCT Types [simplified ASN.1 types]

\[
T := \text{Int } I \quad \text{-- integer type}
\]

\[
| +[T,T,...] \quad \text{-- sum of } T \text{'s}
\]

\[
| \times[T,T,...] \quad \text{-- product (tuple) of } T \text{'s}
\]

\[
| \text{List } I T \quad \text{-- [homogeneous] list of } T
\]

\[
| \text{Bits} \quad \text{-- raw bitstream}
\]

\[
I := \text{Width } n \quad \text{-- } n \text{ bit natural } (0..2^n-1)
\]

\[
| \text{Offset } m I \quad \text{-- integer set offset by } m
\]

\[
| \text{Constrnt } C I \quad \text{-- constrained integers}
\]

\[
C := \text{GTE } m
\]

\[
| \text{EQ } m
\]

\[
| \text{Not } C
\]

\[
| \text{And } C C
\]

\[
m := <\text{signed integer literals}>
\]

\[
n := <\text{natural number literals}>
\]
CMPCT Values

\[ v := m \quad \text{-- integer} \]
\[ \text{Inject}_n v \quad \text{-- sum element} \]
\[ (v, v, \ldots) \quad \text{-- tuple} \]
\[ [v, v, \ldots] \quad \text{-- list} \]
\[ \text{bits} \quad \text{-- raw bitstream value} \]

\[
\text{bits} := (0 \mid 1) \text{bits} \\
\mid <>
\]
Type Level:

\[
\text{Bool} = \text{Int}(\text{Width} \ 1)
\]

A tuple value:

\[
p1 = (100, 1) :: \times[\text{Int}(\text{Width} \ 8), \text{Bool}]
\]

Sums:

-- the constructors for \([A,B]::\):

\[
\text{Inject}_0 :: A -> [A,B] \\
\text{Inject}_1 :: B -> [A,B]
\]

-- values:

\[
a = \ldots :: A \\
b = \ldots :: B \\
\text{sumA} = \text{Inject}_0 a :: [A,B] \\
\text{sumB} = \text{Inject}_1 b :: [A,B]
\]
\begin{verbatim}
b :=
    -- canonical encoder/decoders --
    Int I
    | SumN \([b, b, \ldots]\) -- 2^n elements
    | Seq \([b, b, \ldots]\)
    | ListE \(b \ b\)
    -- bijective operators --
    | inverse \(b\)
    | \(b \cdot b\) -- composition
    | Id \(T\) -- identity bijection
    | prim \(p\)
    -- functors --
    | +(b,b,...)
    | \(\times(b,b,...)\)
    | ListF \(b\)
\end{verbatim}
The type for bijections:

\[ T \leftrightarrow T \]
The type for bijections:

\[ T \iff T \]

An encoder for type \( t \) would have this type:

\[ EE(t) \iff \text{Bits} \]

\( EE(t) \) (Extend with Errors) is the type \( t \) extended with the possible errors that decoding might introduce (including also extraneous bits).

- defined such that encoder/decoders are always bijections.
- details later
CMPCT Bijectons: Examples

Type level:

\[ \text{Bool} = \text{Int}(\text{Width 1}) \]

Bijectons:

\[ \text{w3} = \text{Int}(\text{Width 3}) :: \text{EE}(\text{Int}(\text{Width 3})) \iff \text{Bits} \]
\[ \text{bool} = \text{Int}(\text{Width 1}) :: \text{EE}(\text{Bool}) \iff \text{Bits} \]

\[ \text{ende\_Sum} = \text{SumN}\ [\ w3, \ \text{bool}\ ] \]
\[ :: \text{EE}(+[\text{Int}(\text{Width 3}),\text{Bool}]) \iff \text{Bits} \]

\[ \text{ende\_Pair} = \text{Seq}\ [\ w3, \ \text{bool}\ ] \]
\[ :: \text{EE}(\times[\text{Int}(\text{Width 3}),\text{Bool}]) \iff \text{Bits} \]

(warning: overloading!)
1. Background

2. The Big Idea: Generalizing Encoders and Decoders

3. The CMPCT DSL (Bijections All The Way Down)

4. Structuring 'Bad': the "Extend with Errors" type

5. Expressiveness & Use Cases

6. Conclusion
Defining "Extend with Errors" (EE)

\[ EE(t) = \times \left[ + \left[ t \quad -- \text{successful decode of type } t \\
, \text{ERRORSOF}(t) \quad -- \text{all decoding errors for } t \right] \\
, \text{Bits} \quad -- \text{undecoded bits} \right] \]

EE(t) uniquely identifies the error as well as everything known at the time of error-detection.

- thus we can encode the whole invalid bit stream from EE(t).

ERRORSOF(t) is defined inductively over the type t.
Examples of ERRORSOF

\[
\text{ERRORSOF}(\times[A,B,C]) = \\
\times[\times[\times[A,B,C]]\times\text{ERRORSOF}(A)\times\text{ERRORSOF}(B)\times\text{ERRORSOF}(C)\times\text{ERRORSOF}(D)\times\text{ERRORSOF}(E)]
\]

\[
\text{ERRORSOF}([A,B,C,D]) = \\
\times[A,B,C,D]\times\text{ERRORSOF}(A)\times\text{ERRORSOF}(B)\times\text{ERRORSOF}(C)\times\text{ERRORSOF}(D)
\]

\[
\text{ERRORSOF}(\text{Int}(\text{Width } w)) = \text{Bits} \\
\text{-- incomplete, } \leq w \text{ bits in stream}
\]
Observations re ERRORSOF

- Breaks down into the basic Sum, Prod, Int, Bits types.
- Each of which has a canonical encoding
- Thus $\text{ERRORSOF}(x)$ has a canonical encoding.
  - which conveniently are the bits that elicited the error!
Defining ERRORSOF

Warning: detailed code ahead . . .
### Defining \textsc{ERRORSOF}

\begin{verbatim}
ERRORSOF(×ts) = +[ ×[ ×(take i ts) -- the partial product , ERRORSOF(ts!!i) -- but always ends with last error ]
  | i <- [0..(length ts -1)] ]
ERRORSOF(+ts) = INCOMPLETE len -- not enough bits for tag + +[ERRORSOF(t) | t <- ts] -- error in the sum element
ERRORSOF(Int i) = INT_ERRORS(i)

INT_ERRORS (Width w ) = INCOMPLETE w
INT_ERRORS (Offset o i ) = INT_ERRORS i
INT_ERRORS (Cnstrnt c i) = +[ Int(Cnstrnt (Not c) i) -- constraint failed, return int , INT_ERRORS(i) -- error in the underlying type ]
INCOMPLETE w = Bits -- when the remaining bits in bitstream are fewer than the -- required 'w' bits, holds the "incomplete" bitstream.
\end{verbatim}
1 Background

2 The Big Idea: Generalizing Encoders and Decoders

3 The CMPCT DSL (Bijections All The Way Down)

4 Structuring 'Bad': the "Extend with Errors" type

5 Expressiveness & Use Cases

6 Conclusion
A Question of Expressiveness

We have

- only bijections (not functions, lenses, etc.)
  - small set of primitive bijections
- only one canonical enc/dec for each of Int, Product, Sum, List
  - sums only with $2^n$ elements
A Question of Expressiveness

- We have
  - only bijections (not functions, lenses, etc.)
    - small set of primitive bijections
  - only one canonical enc/dec for each of Int, Product, Sum, List
    - sums only with $2^n$ elements

- Can we really . . .
  - Encode all ASN.1 type constructs into UPER?
  - Encode some of the complicated integer and length encodings?
  - Encode other bit-oriented encoding rules (PER variants . . .)
  - Encode similar byte-oriented encoding rules (OER)?
A Question of Expressiveness

- We have
  - only bijections (not functions, lenses, etc.)
    - small set of primitive bijections
  - only one canonical enc/dec for each of Int, Product, Sum, List
    - sums only with $2^n$ elements

- Can we really . . .
  - Encode all ASN.1 type constructs into UPER?
  - Encode some of the complicated integer and length encodings?
  - Encode other bit-oriented encoding rules (PER variants . . .)
  - Encode similar byte-oriented encoding rules (OER)?

In short, pretty-much (;-)}
Use Cases

(To provide intuition for the expressiveness of CMPCT.)
Use Case 1: Optional in SEQUENCE

This ASN.1:

\[
T3 ::= \text{SEQUENCE} \{ a \ A \ \text{OPTIONAL}, \\
    b \ B \ \text{OPTIONAL}\}
\]

Maps naturally to this CMPCT:

\[
\text{Unit} = () \quad -- \text{name the 0-tuple} \\
\text{Optional}(t) = +[t, \text{Unit}] \quad -- \text{like Maybe in Haskell}
\]

\[
T3 = \times[\text{Optional}(A), \text{Optional}(B)] \\
t3 = \text{Seq} \left[ \text{optional}(a) \\
    , \text{optional}(b) \\
\right] \\
:: \text{EE}(T3) \iff \text{Bits}
\]

- \( t3 \) doesn’t match ASN.1 UPER encoding.
Use Case 1: Optional in SEQUENCE (2)

The canonical encoding of $T3'$ is what matches the ASN.1 encoding

$$T3' = +[\text{Unit, B, A, A} \times \text{B}] \quad \text{-- 4 sum, 2-bit "tag"}$$

We can write this

$$\text{glue} :: \times[\text{Optional}(A), \text{Optional}(B)] \iff +[\text{Unit, B, A, A} \times \text{B}]$$

$$\text{onResult (b :: T1 \iff T2)} :: \text{EE}(T1) \iff \text{EE}(T2)$$

$$t3 = \text{onResult glue . t3'} :: \text{EE}(\times[\text{Optional}(a), \text{Optional}(b)]) \iff \text{Bits}$$
Use Case 1: Optional in SEQUENCE (3)

Optional(A) × Optional(B) 
\[\iff\] \{definition of Optional\} 
(Unit+A) × (Unit+B) 
\[\iff\] \{distl\} 
Unit×(Unit+B) + A×(Unit+B) 
\[\iff\] \{+[distr, distr]\} 
(Unit×Unit + Unit×B) + (A×Unit + A×B) 
\[\iff\] \{sum-flatten -\} 
+[Unit×Unit, Unit×B, A×Unit, A×B] 
\[\iff\] \{+[unitL, unitL, unitR, Id]\} 
+[Unit, B, A, A×B]
Use Case 2: rejectRight and Sums

We sometimes want to look under the hood of $\text{ERRORSOF}(t)$: this is "advanced" usage, but it allows us to turn a valid, decoded value into an error (or the reverse):

\[
\text{rejectRight} :: \times [+[a+b, e], \text{Bits}] \quad \leftrightarrow \quad 'b' \quad \text{a good case}
\]
\[
\times [+[a, b+e], \text{Bits}] \quad \leftrightarrow \quad 'b' \quad \text{an error case}
\]

\text{rejectRight} allows us to encode/decode sums that aren’t of length $2^n$. 
Use Case 3: Numeric Representations

This isomorphism allows us to represent 0 with one bit in an integer type:

\[ \text{Int}(\mathbb{Z} \ n) \iff \text{Int}(\mathbb{Z} \ 1) + \text{Int}(\mathbb{Z} \ (n-1)) \]
The Question of Expressiveness (2)

- Can we really ...
  - Encode all ASN.1 type constructs into UPER?
  - Encode some of the complicated integer and length encodings?
  - Encode other bit-oriented encoding rules (PER variants ...)
  - Encode similar byte-oriented encoding rules (OER)?
The Question of Expressiveness (3)

- Assuming,
  - Ignore recursive types (rare in ASN.1 specs)
  - Add new primitive encoder/decoder for byte-aligned products (needed for OER and PER [the aligned version of UPER])

- Conjecture: CMPCT can express the encoder/decoders needed for
  - all ASN.1 types
  - the bit and octet based encoding rule sets (PER..., OER...)
Elaborating this Conjecture . . .

- CMPCT types are sufficient to capture non-recursive ASN.1 types
  - The simplicity is surprising due to the extreme complexity of ASN.1.

- CMPCT can capture the encoding schemes of interest
  - We appear to have sufficient primitives for low level decoding
  - We can use powerful bijections to "massage" low level things into desired types.
    - N.B., we’ve got a number of useful numeric bijections [paper!]
A Price to Pay?

- non-canonical encodings (non-injective decoders) are handled awkwardly, (the same issue with "don’t cares" in the bit-stream).
- numerous arithmetic bijections under the hood: require some fancy datatypes
- no type inference
Background

The Big Idea: Generalizing Encoders and Decoders

The CMPCT DSL (Bijections All The Way Down)

Structuring 'Bad': the "Extend with Errors" type

Expressiveness & Use Cases

Conclusion
I.e.,

- Generate invalid bit-streams for J2735 that give "good" coverage.
- Can we know we’ve covered every class of error?
Solving The Immediate Need

- I.e.,
  - Generate invalid bit-streams for J2735 that give "good" coverage.
  - Can we know we’ve covered every class of error?

- Yes,
  - a nice definition of "class of error":
    - each sum alternative in $\text{ERRORSOF}(t)$ is a class of error.
    - we’d expect this to correspond to control-flow branches in an implementation.
  - a way to measure "coverage":
    - the number of tests generated for each "class of error"
I.e.,

- A solution that works?
  - for any ASN.1 type (not ad hoc for J2735)
  - for other "encoding rule" schemes besides UPER
Solving The Immediate Need (2)

- I.e.,
  - A solution that works?
    - for any ASN.1 type (not ad hoc for J2735)
    - for other "encoding rule" schemes besides UPER

- Yes,
  - in theory, should work for nearly all ASN.1 types and encoding rules
  - If we change encodings (e.g., switch to OER), an updated generate invalid vectors would come for free.
Contributions

- We’ve captured the Essence of ASN.1 with CMPCT.
  - We can capture ASN.1 types and constraints
  - We can capture multiple *encoding rule sets*
- Shown how (surprisingly) expressive one can get even
  - with limited "dependent types"
  - restricting ourselves to pure bijections
- A principled way to lift encoder/decoder pairs into bijections
  - the EE method for capturing errors.
Questions?