ASN.1 Encoding Schemes Done Right Using CMPCT

Mark Tullsen
Galois, Inc.
tullsen@galois.com

Abstract

Abstract Syntax Notation One (ASN.1) is a ubiquitous data description language for defining data that can be serialized and deserialized across platforms. ASN.1 supports multiple encoding schemes, referred to as "encoding rules". Each encoding rule set specifies how to represent abstract values as a sequence of bits. Some of the more common encoding rules are Basic Encoding Rules (BER), Distinguished Encoding Rules (DER), Packed Encoding Rules (PER), and XML Encoding Rules (XER). In the process of validating the correctness of an ASN.1 encoder/decoder pair for the J2735 standard (Dedicated Short Range Communications Message Set for Vehicle to Vehicle communications), we have designed an intermediate language for describing ASN.1 types as well as ASN.1 encoding schemes. Our intermediate language, *CMPCT*, demonstrates the elegance of using "bidirectional transformation" methods. *CMPCT* allows one to create encoding schemes that are correct by construction.

1 Introduction

This section describes how the need for a high assurance implementation of a vehicle to vehicle (V2V) protocol led to the design of *CMPCT*, our domain specific language (DSL) for describing ASN.1 encoding rule schemes.

1.1 Vehicle to Vehicle (V2V)

Vehicle-to-vehicle (V2V) communication's ability to wirelessly exchange information about the speed and position of surrounding vehicles shows great promise in helping to avoid crashes, ease traffic congestion, and improve the environment.[1] The key message broadcast between vehicles in V2V communications is the Basic Safety Message (BSM). It is standardized under SAE J2735 [DSR16], where it is defined using the ASN.1 data description language. However, this new interface into the automobile introduces a new attack surface into the whole transportation system[2]. We want to ensure that V2V software is robust and secure. One part of achieving this is to ensure that the encoding and decoding of BSMs is secure.

In: J. Cheney, H-S. Ko (eds.): Proceedings of the Eighth International Workshop on Bidirectional Transformations (Bx 2019), Philadelphia, PA, USA, June 4, 2019, published at http://ceur-ws.org

[1]See `https://www.nhtsa.gov/technology-innovation/vehicle-vehicle-communication`

[2]See [CMK+11] for a description of existing attack vectors.

1.2 The ASN.1 Data Description Language and Encoding Rules

ASN.1 is a data description language for specifying data formats and messages. Although it can express relations between request and response messages, it was not designed to specify stateful protocols. ASN.1 was first standardized in 1984, with many revisions since.

While ASN.1 is "just" a data description language, it is quite large and complex. Indeed, merely parsing ASN.1 specifications is difficult. Dubuisson notes that the grammar of ASN.1 (1997 standard) results in nearly 400 shift/reduce errors and over 1,300 reduce/reduce errors in a LALR(1) parser generator, while a LL(k) parser generator results in over 200 production rules beginning with the same lexical token [Dub00]. There is a by-hand transformation of the grammar into an LL(1)-compliant grammar, albeit no formal proof of their equivalence [FPF96].

Not only is the syntax of ASN.1 complex, but so is its semantics. ASN.1 contains a rich data-type language. There are at least 26 base types, including arbitrary integers, arbitrary-precision reals, and 13 kinds of string types). Compound data-types include sum types (e.g., `CHOICE` and `SET`), records (i.e., `SEQUENCE`) with subtyping, and recursive types. There is a complex constraint system (ranges, unions, intersections, etc.) on the types. Subsequent ASN.1 revisions support open types (providing a form of dynamic typing), versioning to support forward/backward compatibility, user-defined constraints, parameterized specifications, and *information objects* which provide an expressive way to describe relations between types.

Beyond the data description language itself, ASN.1 also specifies a number of encoding systems, referred to as *encoding rules*, to describe how ASN.1 abstract data is serialized. There are over a dozen standardized ASN.1 encoding rules. Most rules describe 8-bit byte (octet) encodings, but three rule sets are dedicated to XML encoding. Common encoding rules include the Basic Encoding Rules (BER), Distinguished Encoding Rules (DER), and Packed Encoding Rules (PER). Encoder and decoder pairs are always with respect to a specific type schema and a specific encoding rule set.

1.3 Examples of ASN.1

To provide a concrete flavor of ASN.1, we present an example data *schema*. Let us assume we are defining messages that are sent (TX) and received (RX) in a query-response protocol.

```
MsgTx ::= SEQUENCE {
  txID  INTEGER(1..5),
  txTag UTF8STRING
}
MsgRx ::= SEQUENCE {
  rxID INTEGER(1..7),
  rxTag SEQUENCE(SIZE(0..10)) OF INTEGER
}
```

We have defined two top-level types, each a `SEQUENCE` type. A `SEQUENCE` is an named tuple of fields (like a C struct). The `MsgTx` sequence contains two fields: `txID` and `txTag`. These are typed with built-in ASN.1 types. In the definition of `MsgRx`, the second field, `rxTag`, is the `SEQUENCE OF` type; it is equivalent to an array of integers that can have a length between 0 and 10, inclusively. Note that the `txID` and `rxID` fields are *constrained* integers that fall into the given ranges.

ASN.1 allows us to write values of defined types. The following is a value of type `MsgTx`:

```
msgTx MsgTx ::= {
  txID  1,
  txTag "Some msg"
}
```

1.4 ASN.1 Security

There are currently over 100 vulnerabilities associated with ASN.1 in the MITRE Common Vulnerability Enumeration (CVE) database [MIT17]. These vulnerabilities cover many vendor implementations as well as encoders and decoders embedded in other software libraries (e.g., OpenSSL, Firefox, Chrome, OS X, etc.). The vulnerabilities are often manifested as low-level programming vulnerabilities. A typical class of vulnerabilities are unallowed memory reads/writes, such as buffer overflows and over-reads and NULL-pointer dereferences. ASN.1 was recently featured in the popular press when an ASN.1 vender flaw was found in telecom systems, ranging from cell tower radios to cellphone baseband chips [Goo16]; an exploit could conceivably take down an entire mobile phone network.

Multiple aspects of ASN.1 combine to make ASN.1 implementations a rich source for security vulnerabilities. One reason is that many encode/decode pairs are hand-written and ad-hoc. There are a few reasons for using ad-hoc encoders/decoders: while ASN.1 compilers exist that can generate encoders and decoders, all but the most expensive lack support for full ASN.1 and/or do not support many encoding rules. The only tools that support the full language are proprietary and expensive.

Even if an ASN.1 compiler is used, the compiler will include significant hand-written libraries for encoding and decoding base types, for memory allocation, etc. For example, the unaligned packed encoding rules (UPER) require tedious bit operations to encode types into a compact bit-vector representation. Indeed, a recent vulnerability discovered in telecom systems is not in protocol-specific generated code, but in the associated libraries [Goo16].

Finally, because ASN.1 is regularly used in embedded and performance-critical systems, encoders/decoders are regularly written in unsafe languages, like C. As noted above, many of the critical security vulnerabilities in ASN.1 encoders/decoders are memory safety vulnerabilities in C.

1.5 Creating High Assurance ASN.1 Implementations

Motivated by the risks of potential security vulnerabilities in a V2V decoder implementation, Galois has developed a high assurance implementation of the J2735 Basic Safety Message (BSM) in the C language and has formally verified a large portion of it; This work is described in [TPCT18]. As an adjunct to this work, Galois also has been working on a tool to test and validate other implementations of the standard, in particular to verify that various implementations agree on the messages accepted and rejected.

In order to verify consistency of implementations, our tool must generate test vectors of multiple kinds:

- acceptance test vectors: valid ASN.1 values at the abstract level, i.e., BSM messages represented in C code. We can generate random messages as well as generating random messages near the border cases.
- acceptance test vectors of valid BSM bitstreams. Such vectors can be generated from the previous vectors using our encoder.
- rejection test vectors of *invalid* BSM bit-streams.

Since we have a `decode` function similar to the following[3]:

```
decode :: Bits -> Maybe BSM
```

I.e., it takes `Bits`, the bits on the wire, and returns `Maybe BSM`, i.e., a `Just bsm` if the decoder succeeds and `Nothing` when it fails. We can define a predicate to determine when bitstreams are valid:

```
invalid(bs) = decode(bs)==Nothing
```

and use it to generate bitstreams guaranteed to be invalid:

```
badBitstreams = filter invalid (generateAllBitStreamsUpToLength 600)
```

Alternatively, we might use a `generateRandomBitStreams` function that generates random bitstreams.

Either way, there are problems with this approach: (1) it depends on the `decode` function being correct; (2) we have no idea of the nature of the errors that we are generating, and we have no sense of the "coverage" of the invalid bitstreams; and (3) it is very inefficient (even assuming lazy evaluation as we have in Haskell): we are likely to be generating lots of non-useful `badBitstreams` such as bitstreams containing an "extraneous bits at end" error or generating thousands of different bitstreams all of which cause identical decoder errors (e.g., detected on the 5th bit). It is unclear how one might generate an invalid bitstream in which the error is at the 1000th bit (and no earlier in the bitstream).

Rather than exhaustive or random generation of bitstreams, we would prefer

[3]In our examples, we are using Haskell [HJW92] notation, or sometimes Haskell-like pseudo-code.

- to generate a smaller set of invalid bitstreams; because generally decoding must proceed sequentially, there is rarely a reason to have further bits beyond where the bitstream first becomes invalid.
- for each "logical error"[4] to generate no more than n invalid bitstreams, or alternatively, to generate at least n invalid bitstreams.

In order to create a "better" bitstream generator, we developed a solution that is more general than the immediate need of one ASN.1 spec (J2735.ASN) that uses one ASN.1 encoding rule set (UPER). Our solution, described in what follows, supports most ASN.1 types and many ASN.1 encoding rules (we were focused on the UPER, DER, and OER encoding rules). Our solution

- Works for most ASN.1 types (not supporting recursive types).
- Works for many ASN.1 encoding rules: UPER, PER, OER were in view, but others could be supported.
- Is applicable to other approaches to serialization and deserialization.

2 The *CMPCT* Approach: An Intermediate Language for ASN.1

The goal of this section (2) is to elucidate how *CMPCT* relates to ASN.1; the details and semantics of *CMPCT* will be covered in the following section (3).

2.1 An Intermediate Language for ASN.1

In our approach we compile ASN.1 to an *intermediate representation*, *CMPCT*[5], a language for defining "compact" representations (i.e., bitstreams) of abstract values. (I.e., it defines encoding/decoding pairs.)

Although lower level than ASN.1, *CMPCT* is similar to ASN.1 in a number of ways: *CMPCT* is a typed, declarative language; *CMPCT* can define data types and abstract values in those types; *CMPCT* separates the description of the data from the description of the encoding rules, while supporting multiple encoding rules. Also, *CMPCT* fully supports some of the more complex features of ASN.1 such as integer constraints, intersection, union, ranges, etc.

2.2 An Example

Here is some simple ASN.1 code that defines a type `T1` and a value `v1`.

```
T1 ::= INTEGER(1..5);     -- type definition
v1 T1 ::= 5;              -- value definition
```

The above code looks similar when translated to *CMPCT*:

```
T1 = Range(1,5)    :: TYPE
v1 = 5             :: T1
```

The notation `e::t` indicates that the `e` has type `t`.

2.3 Encoding Systems in ASN.1 and *CMPCT*

In ASN.1 there are fixed number of encoding rule sets (using `r` to range over these), so we would have, for any type `T` the following two functions[6] that are parameterized over both the type `T` and the encoding rules `r`:

```
encode[T][r] :: T -> Bits
decode[T][r] :: Bits -> Maybe T
```

The decode of a bitstream might return an error, thus the Haskell `Maybe` type is being used to indicate this. In ASN.1 there is a fixed and limited set of encoding rule sets (BER, DER, UPER, etc.) and these encoding rules are given informal specifications, as English prose, in the ASN.1 standard documents.

[4] We will formalize "logical error" in section 3.5.

[5] *CMPCT* is not an acronym it is a *compact* form of "compact".

[6] Ignoring for now that some encoding rules on some types are relations, not functions.

In *CMPCT*, the situation is more complicated (though more powerful). First of all we will need to compile the ASN.1 constructs down to *CMPCT*:

```
T1asASN1 = <ASN.1-definition>          :: ASN1
v1asASN1 = <ASN.1-definition>          :: ASN1
T1       = asn1ToCmpctType(T1asASN1)   :: TYPE
v1       = asn1ToCmpctValue(v1asASN1) :: VALUE (having type T1)
```

We will not discuss the `asn1ToCmpctType` and `asn1ToCmpctValue` functions, but these functions remove a significant amount of complexity from the ASN.1 language.

In *CMPCT*, there are no built-in *encoding rule* sets but for each type we have a canonical encoder/decoder. Using other constructs in *CMPCT* we can create arbitrary (of a reasonable nature) encoder/decoders for a given type. Here's how we define an encoder/decoder for a range of integers:

```
T1 = Range(1,5)
encdec_T1 = Int T1  :: ENCDEC(T1)
```

`Int T1` refers to the canonical encoder/decoder for the integer type `T1` (more details are in section 3).

Later we will define `ENCDEC(t)` precisely, but for now one can view `ENCDEC(t)` as the pair of functions for encoding and decoding values of type `t`. So, from `encdec_T1`, we can extract the encoder and decoder:

```
encdec_T1.enc :: Range(1,5) -> Bits
encdec_T1.dec :: Bits -> Maybe (Range(1,5))
```

2.4 ASN.1 and *CMPCT* Compared

To write an ASN.1 compiler using *CMPCT*, one needs functions that specify how to encode the type `t` for each desired encoding rule set, e.g.,

```
encDecForUPER :: (t:TYPE) -> ENCDEC(t)
encDecForDER  :: (t:TYPE) -> ENCDEC(t)
encDecForOER  :: (t:TYPE) -> ENCDEC(t)
```

(In our implementation, these functions would be written in Haskell, and are not considered part of the *CMPCT* DSL per se.) Two things to note here: (1) The language for describing `ENCDEC(t)` is sufficiently powerful to allow these to be written, and (2) the `ENCDEC(t)` values, after *CMPCT* type-checking, will be guaranteed to be a consistent encoder/decoder pair for the type `t`.

CMPCT allows for many other alternatives to the above, for example, on a given type `T1`, we can define various encoding systems:

```
encdec_T1_SCHEME1 = <...>  :: ENCDEC(T1)
encdec_T1_SCHEME2 = <...>  :: ENCDEC(T1)
encdec_T1_SCHEME3 = <...>  :: ENCDEC(T1)
```

There is no implied convertibility between the two encoders `encdec_T1_SCHEME1` and `encdec_T1_SCHEME2`: they both encode the same type, `T1`, but their encodings could be anything.

The key difference between ASN.1 and *CMPCT* is that *CMPCT* can declaratively and unambiguously define an encoding scheme (in ASN.1, the rules are informally described in English prose).

3 The *CMPCT* Domain Specific Language

In this section we define *CMPCT*. *CMPCT* is implemented as a deeply embedded [SA13] domain specific language (DSL) in Haskell. Because *CMPCT* is a deep embedding, we have an interpreter and type-checker for *CMPCT* written in Haskell. (This is in contrast to a shallow embedding in which the interpreter and type-checker would not be needed, we would have "re-used" the Haskell semantics and type-checker.)

3.1 Values and Types

In Figure 1 we show the grammar for the core elements of *CMPCT*. The encodable/decodable values of *CMPCT* are defined by the productions for v (lines 15-22). The types of these values are described by T (lines 1-5).

Types

```
T := Int I         -- integer type
   | +[T,T,...]    -- sum of T's
   | ×[T,T,...]    -- product (tuple) of T's
   | List I T      -- [homogeneous] list of T
   | Bits          -- raw bitstream

I := Width n       -- n bit natural
   | Offset m I    -- integer set offset by m
   | Cnstrnt C I   -- constrained integers

C := GTE m
   | EQ  m
   | Not C
   | And C C
```

Values

```
v    := m             -- integer
      | Inject_n v    -- sum element
      | (v, v, ...)   -- tuple
      | [v, v, ...]   -- list
      | bits          -- raw bitstream value

bits := (0 | 1) bits
      | <>
```

The Bijection Type

```
B := T  ⟺  T
```

Bijections

```
b :=
           -- canonical encoder/decoders --
      Int I
    | SumN [b,b,...] -- 2^n elements
    | Seq [b,b,...]
    | ListE b b
               -- bijective operators --
    | inverse b
    | b . b          -- composition
    | Id T           -- identity bijection
    | prim p
               -- functors --
    | +(b,b,...)
    | ×(b,b,...)
    | ListF b

p := <primitive bijections>
```

...

```
m := <signed integer literals>
n := <natural number literals>
```

Figure 1: The *CMPCT* DSL

We have two primitive types: integers `I` and raw bitstreams `Bits`. Some examples of the integer types `I` are

```
Width 3                                     -- {0..7}, 0 based naturals
Offset 1 (Width 3)                          -- {1..8}
Cnstrnt (Not (GTE 6)) (Offset 1 (Width 3)) -- {1..5}
```

We define the following types to use in following examples:

```
Bool   = Int(Width 1) :: TYPE  -- capture booleans as 1 bit naturals
Word8  = Int(Width 8) :: TYPE
Unit   = ×[]          :: TYPE  -- a product with no sub-elements, cardinality one

Range(m,n) = Int (Cnstrnt (Not (GTE n)) (Offset m (Width w)))
             -- where w is just large enough to hold the range
             -- defines the set of integers {m..n}
Z(n)       = Range(0,n)     -- i.e., the 0 based integers up to n
Optional t = +[t, Unit]     -- like Maybe in Haskell
```

By convention, types are capitalized, while values and bijections are not. In what follows, examples of *CMPCT* code are in Haskell-like pseudo-code and not how it would appear embedded in Haskell. (*CMPCT* is a deep-embedding and we choose not to clutter the examples with the concrete syntax of *CMPCT*.)

We will sometimes use the infix form of the product and sum types: `A+B` for `+[A,B]` and `A×B` for `×[A,B]`.

Here is an example of a value of a product type[7].

```
p1 = (100,1)      :: ×[Word8, Bool]
```

An essential feature in *CMPCT* are *sum* types, an uncommon feature outside of typed functional languages. Sum types can be seen as a generalization of C's *enum* or seen as a disciplined way to tag and safely use C's *union* types. To create an object of a sum type, we use `Inject_n`. When used in the context of a sum type `+[A,B,C]`, the types of the injections are thus:

```
Inject_0 :: A -> +[A,B,C]
Inject_1 :: B -> +[A,B,C]
Inject_2 :: C -> +[A,B,C]
```

and we can use them to create sum values, like so:

```
a = ... :: A
b = ... :: B
sumA = Inject_0 a :: +[A,B,C]
sumB = Inject_1 b :: +[A,B,C]
```

[7] A product is like a C *struct*.

3.2 Encoding Systems as Bijections

To describe an encoding system, *CMPCT* uses *bijections*, as enumerated by the productions for *b* in lines 24-40 of Figure 1. The first bijection `Int i` is the canonical encoding/decoding for the given integer type `i`. E.g.,

```
w3   = Int(Width 3) :: EE(Int(Width 3)  ⟺  Bits
bool = Int(Width 1) :: EE(Bool)    ⟺  Bits
```

A few things to note here:

- The type of bijections always has the form `t1 ⟺ t2`, this signifies that it is a bijection between the two types.
- `EE(t)` represents the type `t` extended with the possible errors that decoding might introduce. It is defined in such a way that we have a bijection; the details will be described in section 3.5.

The three productions in lines 27-29 are the canonical bijections for the encodings of sums, products, and lists. For example,

```
ende_Sum  = SumN [ w3, bool] :: EE(+[Int(Width 3),Bool])  ⟺  Bits
ende_Pair = Seq  [ w3, bool] :: EE(×[Int(Width 3),Bool])  ⟺  Bits
ende_list = List w3 bool     :: EE(List (Width 3) Bool)   ⟺  Bits
```

The `List w3 bool` bijection creates an encoder for a List with `Bool` elements. The length is constrained by `Width 3` (i.e., 0..7), e.g., the decoder first reads 3 bits to determine the length of how many `Bool`-s to decode. Regarding the `SumN` constructor, the number of arguments must be a power of 2 (without loss of generality as will be seen in what follows). *Note well*: there is no primitive for creating a bijection from a pair of inverse functions: we only have primitive bijections and combinators thereon.

In lines 36-38 of Figure 1 we have the sum, product, and list type constructors of *T* (lines 2-4) lifted into *functors* from bijections to bijections. Although we overload the operators, the context should make clear whether the operator is a type constructor (type context) or a functor on bijections (bijection context).

Note that `Unit` is the "identity" for the product (×):

```
unitL :: b  ⟺  Unit × b
unitR :: b  ⟺  b × Unit
```

```
unit = Seq []    :: EE(Unit)  ⟺  Bits
bool = Int Bool  :: EE(Bool)  ⟺  Bits

optional (b :: EE(B)  ⟺  Bits) = SumN [b, unit]
                                :: EE(Optional B)  ⟺  Bits

distl :: (a+b) × c  ⟺  a×c + b×c
distr :: a × (b+c)  ⟺  a×b + a×c    -- derivable from distl

sumPermute p     :: +[t1,t2,t3,...]  ⟺  +(PERMUTE(p,[t1,t2,t3,...]))
productPermute p :: ×[t1,t2,t3,...]  ⟺  ×(PERMUTE(p,[t1,t2,t3,...]))

rank        :: a   ⟺   Z (cardinalityOf a)
divide m    :: Z(m times n)  ⟺  ×[Z m, Z n]
subtract n  :: Z(m plus n)   ⟺  +[Z m, Z n]
```

Figure 2: Bijective Primitives and Built-Ins

A number of primitive and built-in bijections are shown in Figure 2.

Note `divide m` and `subtract n`, these are useful bidirectional arithmetic operators: `m times n` and `m plus n` are type level arithmetic operations. These are needed to encode/decode some of the complicated length encodings in DER and UPER. These arithmetic bijections are borrowed from the author's previous work on a parallel array language [TS16].

3.3 Typing Bijections

Not all values generated by the bijection productions are valid, they must also be typeable in the *CMPCT* type system. If the bijection is typeable, then it will have a bijection type (`T1 ⟺ T2`). A partial and simplified version of the type-system for bijections is shown in Figure 3. We show this primarily to give the reader some intuition for the bijection primitives and combinators.

Primitives for Encoding/Decoding:

```
Int i :: EE(Int i)  ⟺  Bits

a :: EE(A) ⟺ Bits,  b :: EE(B) ⟺ Bits
----------------------------------------
    Seq[a,b] :: EE(×[A,B])  ⟺  Bits

a :: EE(A) ⟺ Bits, b :: EE(B) ⟺ Bits
--------------------------------------
  SumN[a,b] ::  EE(+[A,B])  ⟺  Bits

bi :: EE(Int i) ⟺ Bits, bt :: EE(T) ⟺ Bits
--------------------------------------------
  ListE bi bt ::  EE(List i T)  ⟺  Bits
```

Functors (Bijections to Bijections):

```
  b1 :: T1 ⟺ T2, b2 :: U1 ⟺ U2
-------------------------------------
×[b1,b2] :: ×[T1,U1]  ⟺  ×[T2,U2]

  b1 :: T1 ⟺ T2, b2 :: U1 ⟺ U2
-------------------------------------
+[b1,b2] :: +[T1,U1]  ⟺  +[T2,U2]

            b :: T ⟺ U
-------------------------------------
ListF b :: List i T  ⟺  List i U
```

Various Combinators:

```
Id t :: t ⟺ t

b1 :: A ⟺ B, b2 :: B ⟺ C
---------------------------
    b1 . b2 :: A ⟺ C

    b :: A ⟺ B
--------------------
inverse b :: B ⟺ A
```

Figure 3: Typing Bijections

3.4 Generalizing Encoding Systems to Bijections

The reader may well ask if an encoder/decoder pair can always be represented as a mathematical bijection? There are a couple of reasons why this appears problematic:

First, multiple encodings might be allowable for a given abstract value (i.e., the decoder may not be injective). We might simply not support such encoders: we would still be able to support canonical UPER but most ASN.1 encoding rule schemes would not be expressible in *CMPCT*.

Second, in the presence of invalid bitstreams, clearly `encode` cannot be the inverse of `decode`. One could be tempted to restrict the domain of the `decode` function to only the valid bitstreams; but as discussed above in section 1.5, we want to be able to explore and enumerate the set of invalid bitstreams and we definitely do not want to ignore decode errors.

The key insight in the design of CMPCT *is this: to not restrict* `decode` *but rather to generalize* `encode`*.* This allows us to represent encode/decode pairs as bijections while also turning decode errors into "first class citizens".

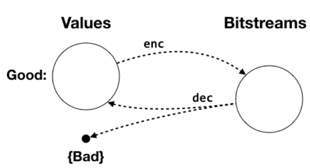

Figure 4

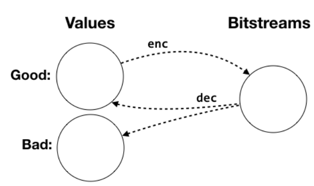

Figure 5

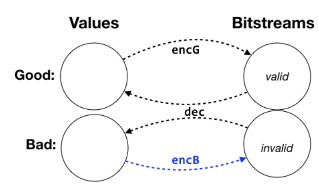

Figure 6

Figure 4 shows the standard scenario where `dec` is the inverse of `enc` and we see effectively *one* error value `Bad`. We can extend `dec` to be injective and thus return a *set* of error values, each one corresponding to the invalid bitstream (this is seen in Figure 5). Next, we extend the encoder to be the pair `encG` (the original encode) and `encB` (the inverse of `dec` restricted to invalid bitstreams). The result can be seen in Figure 6 in which the new, generalized `enc` and `dec` form a bijection and every error value uniquely defines an invalid bit stream.

So, the *generalized* encoder is

```
enc (Good v) = encG v
enc (Bad e)  = encB e
```

and given the new generalized encoder, we can trivially recover the original `enc`:

```
encOriginal v = enc (Good v)
```

and we can recover the original `dec` by throwing away the error "encoding":

```
decOriginal v = case dec bs of
                  Good v -> Good v
                  Bad  e -> Bad ()  -- throw away
```

3.5 Precise Decoding-Errors using *Extended Errors*

Fortunately, the structure of the above `Bad` set can be given a precise structure, based on the type of what we are encoding/decoding. If we are decoding to a value of type `t`, then the result of decode should be `EE(t)` as defined

```
1  EE(t) =
2    ×[ +[ t              -- successful decode, return t
3        , ERRORSOF(t)  -- all possible decoding errors for t
4        ]
5     , Bits            -- undecoded bits following good or bad decode
6     ]
```

That is, we always return the "extra" bits after the decode (line 5). A good decode is indicated with `Inject_0` (line 2), and a decode error is indicated with `Inject_1` (line 3). The type of the decode error `ERRORSOF(t)` is inductively defined over the type of `t` as follows:

```
1  ERRORSOF(×ts)   =
2      +[ ×[ ×(take i ts)    -- the partial product
3          , ERRORSOF(ts!!i) -- but always ends with last error
4          ]
5       | i <- [0..(length ts -1)]
6       ]
7  ERRORSOF(+ts)   = INCOMPLETE len          -- not enough bits for tag
8                 ++[ERRORSOF(t) | t <- ts] -- error in the sum element
9  ERRORSOF(Int i) = INT_ERRORS(i)
10
11 INT_ERRORS (Width w    ) = INCOMPLETE w
12 INT_ERRORS (Offset o i ) = INT_ERRORS i
13 INT_ERRORS (Cnstrnt c i) =
14      +[ Int(Cnstrnt (Not c) i)  -- constraint failed, return integer
15       , INT_ERRORS(i)            -- errors in the underlying type
16       ]
17
18 INCOMPLETE w = Bits
19   -- when the remaining bits in bitstream are fewer than the required
20   -- 'w' bits, holds the "incomplete" bitstream, the length of Bits
21   -- must be less than 'w'
```

Integer errors are pretty straightforward: either there aren't enough bits to encode the integer (`INCOMPLETE`) or a constraint failed. Errors for a sum are pretty straightforward: either we ran out of bitstream before we decoded the tag or we decoded the tag and the associated decoder got an error. Errors for products are a bit more complex (note the use of a Haskell list comprehension in lines 2-6). It is easiest to explain by example:

```
ERRORSOF(×[A,B,C]) =
   +[  ×[]    × ERRORSOF(A)
    ,  ×[A]   × ERRORSOF(B)
    ,  ×[A,B] × ERRORSOF(C)
    ]
```

So, we see that the `ENCDEC(t)` type function used above has a simple definition, and as promised, it is a bijection:

```
ENCDEC(t) =  EE(t)  ⟺  Bits
```

We sometimes want to look inside the `EE` type. Let's redefine `EE` in terms of a helper type function `EE'`:

```
EE(t) = EE'(t,ERRORSOF(t))   :: TYPE
EE'(t,e) = ×[ +[t, e], Bits] :: TYPE
```

Here are some combinators that allow us to manipulate the error side of an `EE`:

```
rejectRight = ×[sumShuffle -, Id Bits]
              :: EE'(a, b+e)  ⟺  EE'(a+b, e)

onResult (b :: T1  ⟺  T2) = ×[+[b, Id -],Id Bits]
                             :: EE'(T1,e)  ⟺  EE'(T2,e)
```

Note the type of `rejectRight`: it moves the type `b` from the "result" type and adds it to the error side (`b+e`). The higher order `onResult` allows us to apply a bijection to the "result" type of an `EE`; it's needed because `T1⟺T2` implies this

```
ERRORSOF(T1)  ⟺  ERRORSOF(T2)
```

but it does not imply that the above two are the same type.

3.6 Non-canonical Encodings

We say an encoding rule set is not "canonical" when there are abstract values that have multiple encodings. (I.e., the decoding function is not injective.) An example would be ASN.1's non-canonical PER. In fact, most encoding rules of ASN.1 are non-canonical.

When a rule set is non-canonical then the encoder will have some choices in how it encodes certain values. The details of such encodings are beyond our scope here, but we can reduce the problem to the following example, motivated by some encoding schemes that give one the encoder a choice whether to encode a "default value" or not encode it. Assume we have desugared the ASN.1 and after decoding we have this:

```
  Unit          -- default value of 6 is unencoded
+ Range(0,6)    -- all values (including 6) can be encoded
```

So,the value `Inject_1 6` is semantically equivalent to `Inject_0 unit`. The decoder clearly needs to support both encodings of 6, so the *CMPCT* approach forces us to generalize the *encoder* to also allow for both encodings of 6 (or a bit to indicate which encoding is desired). This makes the encoder awkward, but we can compose a bijection and get the following:

```
  Range(0,5)
+ Bool        -- the two representations of 6 here
```

We might extend *CMPCT* with mechanisms beyond bijections (using non-bijective lenses or the like) to make creating such encodings more user-friendly.

4 Applications of *CMPCT*

4.1 Use Case: OPTIONAL in SEQUENCE

We have only one type (+) and one canonical encoding for such types (`SumN`), restricted even to 2^n elements. Will this truly be sufficient as a primitive combinator? Especially given all the ways to encode alternatives in ASN.1 such as CHOICE, OPTIONAL, DEFAULT, and etc. We attempt to demonstrate a few examples of the expressiveness of the combinators.

First, we give an example of how to encode the OPTIONAL elements of a SEQUENCE in ASN.1's UPER encoding. We may have the following schema in ASN.1 that indicates that fields `a` and `b` are optional fields:

```
A ::= INTEGER(0..3)    -- needs 2 bits in UPER
B ::= INTEGER(0..7)    -- needs 3 bits in UPER
T3 ::= SEQUENCE {a  A OPTIONAL,
                 b  B OPTIONAL}
```

In UPER is encoded by the concatenation of three bit-fields:

```
[pA,pB]              -- presence bits for the 'a' & 'b' fields
{[a0,a1]     | []}  -- encoding of 'a' when pA is set or nothing
{[b0,b1,b2] | []}  -- encoding of 'b' when pB is set or nothing
```

Now when we translate the above ASN.1 scheme into a *CMPCT* type we get the following:

```
A  = Range(0,3)
B  = Range(0,7)
T3 = ×[Optional(A), Optional(B)]
t3 = Seq [ optional(Int A)
         , optional(Int B)
         ]
     :: ENCDEC(T3)
```

T3 in this *CMPCT* code is the natural representation for the ASN.1 type, but the canonical encoding, defined by t3, gives us a bit encoding for T3; it will be the concatenation of these two bit-fields:

```
{ [0] | [1,a0,a1] }     -- encoding of optional(a)
{ [0] | [1,b0,b1,b2] }  -- encoding of optional(b)
```

So, we have a fundamental mismatch of bits here. However, the following encoding of T3' gives us an encoding that exactly matches the ASN.1 encoding:

```
T3' = +[Unit, B, A, A×B]
t3' =
  SumN [unit                  -- [0,0]
       , Int A                -- [0,1,a0,a1]
       , Int B                -- [1,0,b0,b1,b2]
       , Seq[Int A, Int B]    -- [1,1,a0,a1,b0,b1,b2]
       ]
  :: ENCDEC(T3')
```

Unfortunately T3', although isomorphic to T3, is a bit more awkward to work with. But if we had a bijection between T3 and T3' we would be good to go:

```
glue :: ×[Optional(A), Optional(B)]  ⟺  +[Unit, B, A, A×B]
```

We could then write our desired encoding scheme thus:

```
t3 = onResult glue . t3' :: ENCDEC(×[Optional(a), Optional(b)])
```

We can write glue as a sequence of compositions, viewing it here as this sequence of type isomorphisms:

```
  Optional(A) × Optional(B)
 ⟺                          {definition of Optional}
  (Unit+A) × (Unit+B)
 ⟺                          {distl}
  Unit×(Unit+B) + A×(Unit+B)
 ⟺                          {+[distr, distr]}
  (Unit×Unit + Unit×B) + (A×Unit + A×B)
 ⟺                          {sum-flatten -}
  +[Unit×Unit, Unit×B, A×Unit, A×B]
 ⟺                          {+[unitL, unitL, unitR, Id]}
  +[Unit, B, A, A×B]
```

The sum-flatten bijection has not yet been defined, a scheme for its type is below:

```
sumFlatten -     :: +[+ts,+us,+vs,...]  ⟺  +(ts us vs ...)
productFlatten - :: ×[×ts,×us,×vs,...]  ⟺  ×(ts us vs ...)

sumShuffle -     = sumFlatten -     . sumPermute p     . sumFlatten -
productShuffle - = productFlatten - . productPermute p . productFlatten -
```

(To allow us to gloss over unnecessary details, we write these bijections with placeholders, -, in place of the precise parameters that would make them unambiguous.)

4.2 Bitstream Translations

In section 2 we noted the ability of *CMPCT* to create multiple encoding rules for a single type. As not every bijection needs to be an ENCDEC(t), we can write a translator between two bitstreams (each an encoding of T1 for instance) very simply:

```
translate = encdec_T1_SCHEME1 . inverse encdec_T1_SCHEME2 :: Bits  ⟺  Bits
```

Something to note here: *invalid* bitstreams will be converted back and forth.

4.3 J2735.ASN and Generating Invalid Bitstreams

The motivation for the design of *CMPCT* was to use it to test implementations of the V2V Basic Safety Message (BSM), an ASN.1 UPER encoding. We have used *CMPCT* to encode Part I (the non optional part) of the BSM.

The first segment (Part I) of the BSM is a SEQUENCE with nested SEQUENCES of constrained integer fields containing much of the telemetry data of a vehicle. In essence, it is one large nested product of dozens of constrained integer fields, e.g.,

```
  BSM=
    ×[×[Int F1,Int F2]
     ×[Int F3, ×[Int F4, Int F5]]
     , ...
     ]
  bsm = Seq [ Seq[Int F1,Int F2]
            , Seq[Int F3, Seq[Int F4, Int F5]]
            , ...
            ]
        :: EE(BSM)  ⟺  Bits
        -- or, to expand out EE:
        :: ×[+[BSM,ERRORSOF(BSM)], Bits]   ⟺  Bits
```

We use a QuickCheck like approach [CH00], to generate a good distribution over the type `ERRORSOF(BSM)`, then we inject these values into the type `×[+[BSM,ERRORSOF(BSM)], Bits]`, and then we use our extended encoder to generate "quality" distributions of invalid bitstreams. So, what does `ERRORSOF(BSM)` look like? We'll try to demonstrate this using a small example of a 3-tuple: Using the definition of `ERRORSOF` from section 3.5, we apply it to the type `×[A,B,C]`:

```
1 ERRORSOF(×[A,B,C]) =
2    +[  ×[]    × ERRORSOF(A)
3      ,  ×[A]   × ERRORSOF(B)
4      ,  ×[A,B] × ERRORSOF(C)
5      ]
```

Note a few things: (1) there's only three top-level cases, corresponding to the partial-products of `[A,B,C]`. (2) once we have an error, no more decoding is done. (3) these elements of the Error sum correspond to bitstreams of increasing length. (4) it should be obvious now that we can *encode* the errors, because every error contains all relevant bits that were on the wire.

To generate a desired distribution, we ensure we choose errors from *each* member of the sum of errors and we don't waste test vectors by generating errors after the first error. We don't need to enumerate billions of cases to get to the 50th sum (which we have in the BSM!). Also we don't generate many values for "good" partial tuples: e.g., in line 3 above, we don't need to generate many values for the good partial tuple `×[A]`, but we want to generate most of the possibilities inside `ERRORSOF(B)`.

5 Summary

5.1 Assessments

By generalizing encode/decode pairs to be bijections, we gain a number of advantages:

- The *CMPCT* language becomes simpler, more elegant as all computational elements are bijections. (A previous version of *CMPCT* had a mixture of bijections, encode/decode pairs, and methods to combine the two "universes": this was awkward to implement and use.)
- The language of bijections can be used to write, not just our generalized encoder/decoder pairs (i.e., things of type `EE(t)` $\Longleftrightarrow$ `Bits`) but can all sorts of bijections (i.e., things of type `A` $\Longleftrightarrow$ `B`)
- We guarantee that type-checked bijections in *CMPCT* *are* bijections of the given type. (Whether it is the desired bijection into `Bits` is another matter.)
- With `EE(t)` we have created a typed representation of invalid bit streams in the "abstract value space."

Currently, *CMPCT* allows only finite values: it has no recursion and the `List` type is bounded by an integer range. This limits its expressiveness but it is sufficient for ASN.1 schemes used in practice.

Previous to using *CMPCT*, we had the standard functions

```
enc :: A -> Bits
dec :: Bits -> +[A, Error]
```

and we had the following two functional correctness properties on them.

```
forall a  . dec(enc a) == a
forall bs . case dec(bs) of
              Just x  -> enc x == bs
              Nothing -> True
```

Now that we have a bijection, with these two directions:

```
bsm.enc :: EE(BSM) -> Bits
bsm.dec :: Bits -> EE(BSM)
```

the previous functional correctness properties are simply that we have a bijection:

```
forall a  . bsm.dec(bsm.enc a) == a
forall bs . bsm.enc(bsm.dec(bs)) == bs
```

which is now a property of the correctness of *CMPCT* itself, not about a specific encoder/decoder.

Currently we have implemented the *CMPCT* primitives as described and these are sufficient to encode the ASN.1 "Unaligned PER" encoding scheme (this is a bit-based scheme, the most compact of encoding rules). The *CMPCT* DSL is surprisingly simple, especially when we consider that most of ASN.1 and its octet-based encoding rules can be expressed in *CMPCT*.

Generalizing to "Aligned PER"—in which 0-bits are added at certain points to restore octet alignment—or to other octet based encoding rules would require some extensions to *CMPCT*. E.g., we might add a new primitive of type `ENCDEC(Int t)` which would be octet aligned.

5.2 Related Work

We have been building on our high assurance V2V verification work that has already been described in [TPCT18]. Our test generation capabilities were based on the QuickCheck approach [CH00].

The general research area of bi-directional programming languages is extensive, refer to the survey [CFH+09]. The work on lenses [FGM+07] is also rather extensive. Clearly our work falls into the "bijective lenses" or "bijective language" segment of this field [MFP+17, Fos09]. A large part of the bijective nature of *CMPCT* was inspired and borrowed from the author's previous work [TS16] on a typed functional parallel array language in which we described a language with Sums, Products, Integers and a fixed set of composable bijections.

Some of the design choices of *CMPCT* that have influenced its design, and distinguish it from similar bi-directional languages are the following: static typing, a rich type system with non-trivial computation at the type level, a rich integer constraint system (inherited from ASN.1), combinator based and point free, and lastly, designed not for general computation but for the specific domain of bit-based encoder/decoders.

It may seem surprising that *CMPCT*, being implemented in Haskell, should not use any of the excellent lenses [FPP08] or bijection packages that are available in Haskell[8]. This is due to the mismatch between the *CMPCT* type system and the Haskell type system. We wanted a type system for *CMPCT* which was not at all straightforward to embed into Haskell's type system. The lesson for DSL design is that creating a novel "type-heavy" DSL such as *CMPCT* will benefit from a *deep* embedding (as opposed to the easier to implement *shallow* embedding). Refer to the paper by Hudak [Hud96] on DSLs and refer to [SA13] for a discussion on shallow and deep embeddings. Our ability to rapidly develop multiple iterations of the *CMPCT* DSL was aided greatly by the choice to use a deep embedding.

Now with the design and implementation of core *CMPCT* done, an area to investigate in the future is how easily *CMPCT* could be written in a more general purpose bidirectional language such as PADS [FW11], HOBiT [MW18], or Boomerang [BFP+08].

[8] `https://hackage.haskell.org/package/lens`

5.3 Conclusion

We have described our intermediate language *CMPCT* for describing ASN.1 encoding rule schemes. We have found it useful for our needs in efficiently generating useful rejection tests for J2735 Basic Safety Messages.

CMPCT is similar to ASN.1 in a number of ways: *CMPCT* is a typed, declarative language; *CMPCT* can define data types and abstract values in those types; *CMPCT* separates the description of the data from the description of the encoding rules, while supporting multiple encoding rules. Also, *CMPCT* fully supports some of the more complex features of ASN.1 such as integer constraints, intersection, union, ranges, etc.

However, *CMPCT* improves on ASN.1 in many ways: *CMPCT* can declaratively and unambiguously define an encoding rule scheme (in ASN.1, the encoding rules are described informally); *CMPCT* allows the user to create custom encoding rule schemes, in which the encoder and decoder are guaranteed inverses, by construction; and, while ASN.1 is notorious for a large number of baroque constructs that can interact in unfortunate ways, *CMPCT* has a small set of primitives that can be used compositionally to construct encoding rule schemes.

Acknowledgments.

Much of this work was performed under subcontract to Battelle Memorial Institute for the National Highway Traffic Safety Administration (NHTSA). We thank Arthur Carter at NHTSA for his input and guidance. Our findings and opinions do not necessarily represent those of Battelle or the United States Government.

References

[BFP+08] Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: Resourceful lenses for string data. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 407–419, New York, NY, USA, 2008. ACM.

[CFH+09] Krzysztof Czarnecki, Nate Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James Terwilliger. Bidirectional transformations: A cross-discipline perspective. volume 5563, pages 260–283, 06 2009.

[CH00] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *ACM SIGPLAN Notices*, 35(9):268–279, 2000.

[CMK+11] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, and Tadayoshi Kohno. Comprehensive experimental analyses of automotive attack surfaces. In *USENIX Security*, 2011.

[DSR16] DSRC Technical Committee. Dedicated Short Range Communications (DSRC) message set dictionary (j2735_20103). Technical report, SAE International, 2016.

[Dub00] Oliver Dubuisson. *ASN.1 Communication between heterogeneous Systems*. Elsevier-Morgan Kaufmann, 2000.

[FGM+07] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3), May 2007.

[Fos09] John Nathan Foster. *Bidirectional Programming Languages*. PhD thesis, Philadelphia, PA, USA, 2009. AAI3405376.

[FPF96] Dubuisson (O.) Fouquart (P.) and Duwez (F.). Une analyse syntaxique d'ASN.1:1994. Technical Report Internal Report RP/LAA/EIA/83, France Télécom R&D, March 1996.

[FPP08] J. Nathan Foster, Alexandre Pilkiewicz, and Benjamin C. Pierce. Quotient lenses. *SIGPLAN Not.*, 43(9):383–396, September 2008.

[FW11] Kathleen Fisher and David Walker. The pads project: An overview. In *Proceedings of the 14th International Conference on Database Theory*, ICDT '11, pages 11–17, New York, NY, USA, 2011. ACM.

[Goo16] Dan Goodin. Software flaw puts mobile phones and networks at risk of complete takeover. *Ars Technica*, 2016.

[HJW92] P. Hudak, S. P. Jones, and P. Wadler. Report on the programming language Haskell. 27(5), May 1992.

[Hud96] Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es), December 1996.

[MFP+17] Anders Miltner, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. Synthesizing bijective lenses. *Proc. ACM Program. Lang.*, 2(POPL):1:1–1:30, December 2017.

[MIT17] MITRE. Common vulnerabilities and exposures for ASN.1. Website, February 2017. Available at `https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=ASN.1`.

[MW18] Kazutaka Matsuda and Meng Wang. Hobit: Programming lenses without using lens combinators. In Amal Ahmed, editor, *Programming Languages and Systems*, pages 31–59, Cham, 2018. Springer International Publishing.

[SA13] Josef Svenningsson and Emil Axelsson. Combining deep and shallow embedding for edsl. In *Proceedings of the 2012 Conference on Trends in Functional Programming - Volume 7829*, TFP 2012, pages 21–36, New York, NY, USA, 2013. Springer-Verlag New York, Inc.

[TPCT18] Mark Tullsen, Lee Pike, Nathan Collins, and Aaron Tomb. Formal verification of a vehicle-to-vehicle (V2V) messaging system. In *Computer Aided Verification - 30th International Conference, CAV*. Springer, 2018.

[TS16] M. Tullsen and M. Sottile. Array types for a graph processing language. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 857–866, May 2016.