Formal verification of floating-point number conversion between ASN.1 BER and IEEE 754 binary encodings

Ilia Zaichuk
Taras Shevchenko National University of Kyiv
Digamma.ai
zoickx@knu.ua

Abstract

ASN.1 encoding is widely used for data transfer between various computing systems. Such data may contain floating-point numbers. The most common representation of floating-point numbers is the IEEE 754 standard. ASN.1, however, does not rely on IEEE for that task. Conversion between the two floating-point representations is error-prone in most ASN.1 protocol implementations. In this project, we formalize such conversion and prove its correctness using the Coq proof assistant. Such formalization could be used as an "executable specification" which could be extracted as an OCaml or Haskell program. The secondary goal of this project is to assess the amount of work required for formalizing the full ASN.1 stack, which we are considering as the next step.

1 Problem and motivation

At the fundamental level, any floating-point number is a pair of integers: a *mantissa* (sometimes also called *significand*) and an *exponent*. The main difference between standards for floating-point representation is how this pair is encoded. Different binary lengths might be used, or different representations for special values might be chosen among other variances. These variations in encoding are adapted for assorted use-cases but create complications when conversion from one format to another is required.

An instance of this, which is handled in our project, is the conversion between floating-point numbers, encoded in IEEE 754, and ASN.1 formats. It is a common task but not a simple one. Mere numbers: while any IEEE encoding consists of three different parts, an ASN.1 float has at least five.

To give you an example, in the *asn1c* [Walkin 2017] project, the function converting between the two formats takes around 200 lines of C code. It is quite hard for a human to verify, making a formalization of the topic a much-needed endeavor.

2 Background

In the scope of this project, two formats are considered: the IEEE 754 binary interchange floating-point encoding [597 2008] and the ASN.1 BER real value encoding [ISO 8825-1:2015 2015]. IEEE 754 defines formats of varying lengths, *binary32* and *binary64* being the most commonly used. The IEEE 754 standard has been formalized in Coq by the Flocq [Boldo and Melquiond 2011] library.

Any IEEE 754 floating-point number can be thought of as a triplet: (sign, exp, mantissa) $\in \mathbb{B}\times\mathbb{Z}\times\mathbb{N}$, which is serialized as a concatenation of bit sequences representing each of the components. The core part of any ASN.1 real encoding has the following form: (sign, radix, scaling, exp, mantissa) $\in \mathbb{B}\times\mathbb{N}\times\mathbb{N}\times\mathbb{Z}\times\mathbb{N}$. The serialization scheme for this is more complicated, using additional information about the encoding variants.

Both standards define more than one encoding scheme, but for the purposes of this project, only binary formats are considered: those, in which the exponent uses a base which is a power of two. In addition to real values, the standards support the same set of special values. However, an IEEE *NaN* value loses its debugging payload when conversion to ASN.1 is performed.

While consisting of blocks of similar meaning, ASN.1 and IEEE 754 represent the same numbers in different ways. For example, the number $0.46875 = 15 * 2^{-5}$ corresponds to the $(0, 125, 7340032)$ tuple in IEEE binary32 and $(0, 16, 2, 254, 29)$ in ASN.1 real encoding.

It is important to note that any number in *binary64* or *binary32* IEEE format can be converted to ASN.1 without loss of precision. Conversely, an ASN.1-encoded number is not necessarily representable in any common IEEE format. For such cases, rounding is implemented, as discussed in Section 3.

3 Approach and Uniqueness

In this project, we formalize the conversion between the IEEE 754 and the ASN.1 real encoding standards and assert its correctness using the Coq proof assistant. To our knowledge, this is currently the only project attempting to formalize any part of the ASN.1 standard.

The conversion path is shown in Figure 1. We use $\mathbb{Z}$ to represent bit strings of arbitrary length. Both BER and IEEE floating-point numbers are ultimately encoded as such strings. The conversion is split into several steps with intermediate representations. On the IEEE side, there is *binary_float*, defined in the Flocq library, which provides an

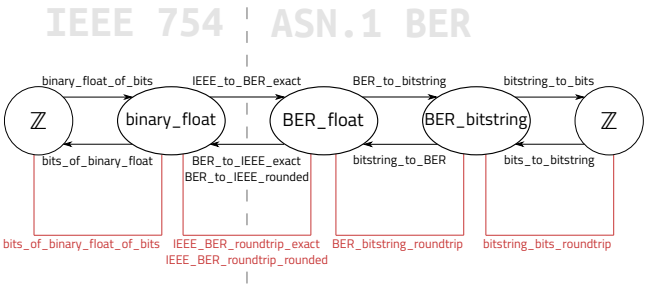

Figure 1. Full conversion scheme

abstract representation of floats as an inductive type with constructors for special values as well as "regular" numbers expressed as a mantissa-exponent pair. On ASN.1 side, there is the *BER_float* inductive type, which is similar to *binary_float*, with some additional parameters unique to BER added, and the *BER_bitstring* type, which is used as an intermediate type for generating sequences of bits.

For each adjacent pair of formats, we need to define two conversion functions for both directions, a heterogeneous equality and lemmas ensuring equivalence of conversion results.

In general, our conversion function from type T to U is not total. The converter may fail for some values of T which could not be represented in U. To express this, our converters have the type $T \longrightarrow \text{option}\, U$, using the *Option Monad*. In case of error, *None* is returned. This allows us to define a generic "round-trip" conversion correctness property as follows:

```
Definition roundtrip
           (A1 B A2 : Type)
           (f : A1 → option B) (* forward pass *)
           (b : B → option A2) (* backward pass *)
           (e : A1 → A2 → 𝔹) (* equality *)
           (x : A1)            (* value *)
  : Prop :=
    is_Some_b (f x) = true →
    𝔹_het_inverse option A1 B A2 f b e x = Some true.
```

The original expression of type $A1$ is converted using "forward" conversion pass to type B and then back using "backward" pass to type $A1$. For example, a practical scenario for this could be encoding (f) a *binary32* ($A1$) float in BER (B), transmitting it over a network, and decoding (b) into *binary64* ($A2$). The results of successful forward conversion must satisfy a heterogeneous equality predicate e, which is applied using the *bool_het_inverse* wrapper:

```
Definition bool_het_inverse
  (m : Type → Type) {Monad m}
  (A1 B A2 : Type)
  (f : A1 → m B)  (b : B → m A2)
  (e : A1 → A2 → 𝔹)(x : A1) : m 𝔹 :=
  y ← f x ;;
    x' ← b y ;;
      ret (e x x').
```

It should be noted that, while we call it a "round-trip," the result of the backwards pass ($A2$) does not have the same type as the original value ($A1$). The reason for this is the need to convert between different formats of IEEE 754. For example, a *binary64* floating-point number might be transmitted using BER format to a different machine where it needs to be represented in *binary32*. In such cases, rounding might need to be performed during conversion from BER to IEEE. This is supported in our implementation, with the roundtrip conversion behaving exactly as if the original number was converted to a different format according to the rules set out in IEEE 754.

For the common IEEE 754 binary formats, high-level encoding and decoding functions are defined. For example for *binary64*:

```
Definition float64_to_BER_exact (f : Z) : option Z :=
  ab ← b64_to_BER_abstract (b64_of_bits f) ;;
    ret (BER_to_bits ab).

Definition BER_to_float64_exact (asn : Z) : option Z :=
  bf ← bits_to_BER asn ;;
    af ← BER_to_b64_abstract_exact bf ;;
      ret (bits_of_b64 af).

Definition BER_to_float64_rounded
  (rounding : mode ) (asn : Z) : option Z :=
    bf ← bits_to_BER asn ;;
      af ← BER_to_b64_abstract_rounded rounding bf ;;
        ret (bits_of_b64 af).
```

The implementation is straightforward. It proceeds by converting a bit string in the source format into the target through intermediate steps corresponding to intermediate formats previously introduced in this paper. If an error occurs at any step (*None* returned), it is propagated to the final result through monadic composition of the steps.

4 Results and Future work

To demonstrate how our verified implementation could be used in a practical ASN.1 encoding/decoding pipeline, we exported our encoder and decoder implementation in Gallina as an OCaml program using Coq *extraction* [Letouzey 2008] facility.

The resulting implementation was then linked to the asn1c library, replacing native implementation of the floating point encoding module. It then passed all unit tests. The performance of the extracted code is, as expected, slower compared to the original C implementation.

To combine benefits of formal verification with high performance of native implementation, we plan to try to verify existing C implementation as the next step using our formalization as the specification. We are considering using VST [Appel 2011] and the CompCert compiler [Leroy et al. 2012].

References

2008. *IEEE Standard for Floating-Point Arithmetic - Redline*. Standard. https://doi.org/10.1109/IEEESTD.2008.5976968

Andrew W Appel. 2011. Verified software toolchain. In *European Symposium on Programming*. Springer, 1–17.

Sylvie Boldo and Guillaume Melquiond. 2011. Flocq: A unified library for proving floating-point algorithms in Coq. In *Computer Arithmetic (ARITH), 2011 20th IEEE Symposium on*. IEEE, 243–252.

ISO 8825-1:2015 2015. *Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)*. Standard. International Organization for Standardization, Geneva, CH. https://www.iso.org/standard/68345.html

Xavier Leroy et al. 2012. The CompCert verified compiler. *Documentation and user's manual. INRIA Paris-Rocquencourt* (2012).

Pierre Letouzey. 2008. Extraction in coq: An overview. In *Conference on Computability in Europe*. Springer, 359–369.

The Coq Development Team. 2018. The Coq Proof Assistant, version 8.8.0. (Apr 2018). https://doi.org/10.5281/zenodo.1219885

Lev Walkin. 2003-2017. ASN1C: The ASN.1 Compiler. https://github.com/vlm/asn1c.