AERES: Summary of verification architecture

Chris Jenkins

December 2, 2021

1 Specification

This section describes the discipline used in Aeres to model the supported subset of X.509. Parsers in Aeres construct values from inputs of type List UInt8 — that is, lists of nonnegative integers which are no greater than 255. Unlike the other X.509 parsers we have discussed, the types of the values constructed from bytestring inputs *depends* upon those inputs. This allows us to have an X.509 parser that is sound *by construction*, because the parser returns a proof that *the input conforms to the specification*.

To illustrate this point, we now look at some Agda definitions that capture a handful of the ANS.1 DER type specifications.

1.1 Agda X.509 Definitions

1.1.1 Short lengths

Consider the definition short lengths, given as Short below.

```
module Length where
  record Short (@erased bs : List UInt8) : Type where
    constructor mkShort
    field
      l : UInt8
      @erased l<128 : toℕ l < 128
      @erased serial : bs ≡ [ l ]
```

- Short is not a type, but a *predicate* (a family of types indexed by bytestrings).

- *bs* is a bound variable, the formal argument to the predicate we are defining. It can also be thought of as a *serialization* of the type we are defining (our modeling discipline allows for the possibility that there may be multiple serializations of a given value; in the case of Short, it is clear that this is not so). It is marked as being erased at runtime with the @erased annotation (the parsed structure does not carry its serialization around).

- mkShort is the constructor for Short *bs* (for all *bs*), requiring three arguments whose types are given by the fields below
  - l is a single byte, whose value is to be interpreted as the length of some other content.
  - l<128 is a constraint on l. Per the ANS.1 grammar, lengths exceeding 127 must be represented by a *long length*.
  - serial relates the data field l to the serialization *bs*. In this case, bs must be equal to the singleton list containing l.

A parser which returns Short *bs* is therefore a sound parser for short lengths *by construction*, because in order to create a value of type Short *bs* one must produce witnesses that *bs* is a singleton list whose sole element is strictly less than 128.

1.1.2 Object Subidentifiers

Here is another example: object sub-identifiers

```
OIDLeastBytes : List Dig → Set
OIDLeastBytes [] = ⊤
OIDLeastBytes (b :: bs) = toℕ b > 128
record OIDSub (@erased bs : List Dig) : Set where
  constructor mkOIDSub
  field
    l_p : List Dig
    l_e : Dig
    @erased l_p≥128 : All (λ d → toℕ d ≥ 128) l_p
    @erased l_e<128 : toℕ l_e < 128
    @erased leastBytes : OIDLeastBytes l_p
    @erased serial : bs ≡ l_p ++ [ l_e ]
```

- The data fields are l_p (the prefix bytes) and l_e (the ending byte).
- Per section 8.19.2 of the X.690 specification,

  > *Each sub-identifier is represented as a series of (one or more) octets. Bit 8 of each octet indicates whether it is the last in the series: bit 8 of the last octet is zero; bit 8 of each preceding octet is one.*

  Therefore, all of the bytes in the prefix must be at least as large as 128, and the ending byte must be strictly less than 128. This is enforced by $\mathsf{l}_p{\geq}\mathsf{128}$ and $\mathsf{l}_e{>}\mathsf{128}$.
- Finally, as X.509 uses DER, we require that the least number of bytes are used for representing the object sub-identifier. The definition of this property is OIDLeastBytes, and the requirement that this holds of the prefix l_p is enforced by leastBytes.

1.1.3 Generic "sequence-of" type

The next Agda definitions model sequences of a single type with a variable number of entries.

```
mutual
  data SeqElems (@erased A : List Dig → Set) : @erased List Dig → Set where
    [] : (@erased bs : List Dig) → SeqElems A bs
    cons : (@erased bs : List Dig) → SeqElemFields A bs → SeqElems A bs
  record SeqElemFields (@erased A : List Dig → Set) (@erased bs : List Dig) : Set where
    inductive
    constructor mkSeqElems
    field
      @erased bs₁ bs₂ : List Dig
      h : A bs₁
      t : SeqElems A bs₂
      @erased serial : bs ≡ bs₁ ++ bs₂
```

- The algebraic datatype SeqElems is defined mutually with the record SeqElemFields.
- SeqElems follows the usual format for defining a list as an algebraic datatype, except that the parameter A gives the type *family* of elements, and the list is indexed by its serialization.

- One constructs a nonempty list with cons, which takes as an argument an element of the record type SeqElemFields.
- Since SeqElemFields is recursive (indirectly, by the type of t), Agda requires we declare it either inductive or coinductive.
- The record has two fields, the head of the list h and the tail t.
- The serialization is the concatenation of the serializations of the head and tail elements (as given by the type of serial).

1.2 Contribution

One of the contributes of this work is a formalization in Agda of a subset of X.509 and X.690 (DER). We believe that this specification is human-readable (though it may require learning some of Agda's notational convention), while at the same time completely unambiguous (compared to the natural language description of both specs).

2 Parsing

The results of a successful parse of a structure `A` from `xs` are given by the `Success` record.

```
record Success (@erased A : List Dig → Set) (@erased xs : List Dig) : Set where
  constructor success
  field
    @erased prefix : List Dig
    read   : ℕ
    @erased read≡ : read ≡ length prefix
    value  : A prefix
    suffix : List Dig
    @erased ps≡ : prefix ++ suffix ≡ xs
```

The unerased fields are the returned values, and the erased fields are part of the specification.

- `prefix`, the bytes consumed during parsing

  This is not returned at run time, but it is needed to state the type of the parsed result.

- `read`, the number of bytes read (enforced by the field `read≡`)

- `value`, the construction of the structure `A` from `prefix`
- `suffix`, the remaining bytes to consume for future parsing.
- The field `ps≡` guarantees that `prefix` and `suffix` are correctly named.

With the definition of the type of results of parsing, we define a parser itself as a record wrapping a function from byte strings to "possible `Success` es" — the structure `Parser` is parameterized by a type constructor `M` to allow for flexible handling of failure.

```
record Parserᵢ (M : List Dig → Set → Set) (A : List Dig → Set) : Set where
  constructor mkParser
  field
    runParser : (xs : List Dig) → M xs (Success A xs)
open Parserᵢ public

Parser : (M : Set → Set) (A : List Dig → Set) → Set
Parser M = Parserᵢ (const M)
```

For parsing X.509, the environment `M` for failure will always involve `Dec`, discussed next. (TODO: `Dec` should probably be part of the definition of `Parser`$_i$)

2.1 `Dec` and complete parsing

In the Agda standard library, the type `Dec : Set -> Set` is the type of "decisions" about a proposition `P : Set`. That is, a proof of `Dec P` is either a proof of `P` or a proof of `¬ P`.

```
data Dec (P : Set) : Set where
  yes :   P → Dec P
  no  : ¬ P → Dec P
```

Now consider the use of `Dec` in the context of parsing

```
parseInt : Parser Dec Int
```

where `Int` is the X.690 DER encoding of an integer. When this parser is run on a byte string `xs`, it returns `Dec (Success Int xs)`. There are two options.

- if the decision is `yes`, then we have a proof that there exists some prefix of `xs` which conforms to the specification `Int`
- if the decision is `no`, then we have a proof that **no prefix exists** from which an `Int` may be parse

Because the parser returns a *decision* on whether a successful parse is possible, we have completeness as well as soundness. Consider the following proof.

```
record ⊤ : Set where
  constructor tt

data ⊥ : Set where

True : ∀ {P} → Dec P → Set
True (yes _) = ⊤
True (no  _) = ⊥

completeness : ∀ {bs} → Success Int bs → True (runParser parseInt bs)
completeness{bs} cert
  with runParser parseInt bs
... | (yes _) = tt
... | no ¬cert = contradiction cert ¬cert
```

- ⊤ is a trivially inhabited type (a true proposition)
- ⊥ is a trivially uninhabited type (a false proposition)
- `True` computes a type by case analysis on a decision over some proposition. It is defined in such a way that a term of type `True (runParser parseInt bs)` implies that `runParser parseInt bs` was successful
- In the proof, we assume that an oracle has given us a successful parse of an `Int` from byte string `bs`. We show that this means the parser *must* succeed as well.
  - Of course, if the parser does succeed (the `yes` case), then we are done — the goal is `True (yes _)`, or more succinctly ⊤.
  - If we fail, the parser returns a proof that there is *no* way to parse an `Int` from `bs`, contradicting our assumption.

3 Lemmas

The proof effort makes use of several lemmas concerning the specification[1], which on their own may be seen as a minor contribution about the properties of the X.690 and X.509 languages.

- `NonNesting` (should be: `Unambiguous`)

  The property that there is only one way to parse a structure `A` from a given byte string

  ```
  NonNesting : (A : List Dig → Set) → Set
  NonNesting A = ∀ {xs₁ ys₁ xs₂ ys₂} → xs₁ ++ ys₁ ≡ xs₂ ++ ys₂
                 → A xs₁ → A xs₂ → xs₁ ≡ xs₂
  ```

  In particular, it is relatively easy to show that TLV structures are unambiguous, because the length of the content is encoded in the byte string itself.

- `Unambiguous` (should be: `Unique`)

  Byte strings uniquely determine the fields of the structure.

  ```
  Unambiguous : (A : List Dig → Set) → Set
  Unambiguous A = ∀ {xs} → (a₁ a₂ : A xs) → a₁ ≡ a₂
  ```

- `NoConfusion`

  It is not possible to confuse the structure `A` for the structure `B` when parsing a byte string; put another way, it is not possible to both be able to parse an `A` and a `B` from the same byte string. This is needed when e.g. some fields are optional.

  ```
  NoConfusion : (A B : List Dig → Set) → Set
  NoConfusion A B = ∀ {xs₁ ys₁ xs₂ ys₂} → xs₁ ++ ys₁ ≡ xs₂ ++ ys₂
                    → (A xs₁ → ¬ B xs₂)
  ```

[1]These may not need to be mentioned in the paper, but I will describe them for the sake of completeness

3.1 Parser Combinators

At present, Aeres is intended to be used as a stand-alone application for differential testing of X.509 parser implementations. However, the development contains several generic parser combinators that could be packaged into their own library for sound and complete parsing of arbitrary languages.

For example, here is the type of a parser combinator which takes a parser for `A` and returns a parser for `A` in which the length of the byte string read is exactly `n`

```
ExactLength : (@erased A : List Dig → Set) → ℕ → @erased List Dig → Set
ExactLength A n xs = A xs × Erased (length xs ≤ n)

parseExactLength : {@erased A : List Dig → Set} → @erased NonNesting A →
                   Parser Dec A →
                   ∀ n → Parser Dec (ExactLength A n)
```

where `Erased` is a record containing a single field of the given type, erased at run time.

```
record Erased (@erased A : Set) : Set where
  constructor mkErased
  field
    @erased x : A
```

Here, we require as an assumption that `A` is `NonNesting` (that is, has no left recursion). If we did not have this, then there may be multiple ways to parse `A` from a given byte string. If the given parser succeeds but returns a `A` built from a prefix that is not the specified length, we would not be able to conclude that there is **no** way to parse `A` from that byte string such that we consume exactly `n` bytes.