A formal semantics for ASN.1

Paul Steckler
Galois
steck@galois.com
What is ASN.1?

- A data description language
- Describes the structure of data to be transmitted over wires (cf. XML schemas)
- Conventional collection of primitive data types: booleans, integers, strings, time types; plus enumerations, records, sum types
- Choice of several encoding schemes
- Specifications can span several modules
- Modules can be mutually-referential
ASN.1 is everywhere

- Many IETF RFCs
- X.509, SNMP, X.400, X.500
- SSL/TLS
- Code for ASN.1 types in every OS, browser
Example ASN.1 module

MyModule
DEFINITIONS ::= 
BEGIN
  EXPORTS ALL;
  IMPORTS;

  T0 ::= [1] INTEGER
  x T0 ::= 42

  T2 ::= [2] BIT STRING { a(1), b(x), c(3) }
  v2 T2 ::= c

END
Vision: High Assurance ASN.1 Workbench

Platform-Independent Protocol Messages

Design

Specify

Validate

ASN.1 Interpreter

CMS, CMC, X.509, X.501...

Assured Implementation

Test cases/harnesses

Validate ASN.1 implementations

Model checking

C or Java

Verify ASN.1 implementations

Generate ASN.1 implementations

ASN.1 Tools
Why a formal semantics?

- Except for the grammar defining the syntax, ASN.1 is specified entirely in English
- The ITU X.680 spec is mostly about syntax, not semantics
- Some of the subtleties are explained using examples in Annexes - not dispositive
- There's no reference implementation
- Potential for error if different compilers used for encoder and decoder
What to do with the semantics?

- Determine which ASN.1 specifications are legal
- If not legal, why not
- Give a meaning for a legal specification
- Exposes subtleties and ambiguities
Who wants a semantics?

- Tool implementers
- Users of ASN.1 tools
- ASN.1 specification writers
- ASN.1 standards writers
- Galois
  - Proof-of-concept compiler
  - Interpreter
  - Verifying compiler
What kind of semantics?

- Denotational semantics: mapping from source syntax to well-understood mathematical meaning
  - Meaning of a syntax phrase is compositional in the meaning of its subphrases
- In an ASN.1 specification, the interesting phrases are type assignments, like

  \[ T1 ::= \text{INTEGER} \{ x(42) \} \]

- And value assignments:

  \[ v1 \ T1 ::= 5280 \]
What are the denotations?

• An ASN.1 compiler generates an encoder and decoder for each defined type

• So the semantics associates encoders and decoders with the types in type assignments
Compositionality of denotations

- Meaning of aggregate types, such as SEQUENCE, depends on the meaning of their components.
- Meaning of a module is the union of the meanings for each type and value defined, producing type and value environments for the module.
- Meaning of a set of modules is the union of the meaning of the modules, yielding global type and value environments.
Formal semantics: precedents

- R5RS, the last-published standard for Scheme, contained a denotational semantics for the lambda-calculus core.
- The Standard ML programming language has had two versions of a formal semantics (1990, revised in 1997)
  - The ML Kit started as a direct implementation of the formal semantics.
  - Compiler implementers can use the Kit as a check on their work, and a vehicle for experimentation.
The semantics covers a subset of ASN.1:

- X.680 only; no parameterization, no information objects, no general constraints
- No extensibility for enumerations, SEQUENCE, etc.
- No XML
- Supported types: BOOLEAN, INTEGER, ENUMERATED, BIT STRING, OCTET STRING, NULL, SEQUENCE/OF, SET/OF, CHOICE, OBJECT IDENTIFIER, RELATIVE-OID, most strings, time types
- Constraints: single value, range, size
The rest of the talk

- What does the semantics look like?
- How the semantics handles encoding rules
- Ambiguities and infelicities
- Type and value compatibility
- Status
Denotations in code

- ASN.1 syntax maps to Haskell expressions
  - An executable specification!
- We already have a representation of ASN.1 syntax from proof-of-concept compiler; some other recycled code
- Advantage of Haskell: the type system documents our logic and checks our work
- Meaning of a type assignment is an encoder / decoder pair, i.e., a pair of Haskell functions (plus some other administrative data)
Semantics for BOOLEAN

\[ mk_{en\_de\_bool} :: MkEnDe \]
\[ mk_{en\_de\_bool} = MkEnDe \circ pairFuns mk_{en\_bool} mk_{de\_bool} \]
where
\[ mk_{en\_bool} \text{ tags} = Encoder \circ \]
\[ \langle \text{ASN1Boolean } b \rangle \rightarrow \]
\[ \text{DataStream } [(\text{tags},\text{PrimDatum } \circ \text{PrimBool } b)] \]
\[ mk_{de\_bool} \text{ tags} = Decoder \circ \]
\[ \langle \text{ds} \rightarrow \text{case headDataStream } ds \text{ of} \]
\[ (\text{tags}',\text{PrimDatum } (\text{PrimBool } b)) \mid \text{tags} == \text{tags}' \]
\[ \rightarrow \text{Just } (\text{ASN1Boolean } b,\text{tailDataStream } ds) \]
\[ _ \rightarrow \text{Nothing} \]
Semantics for SEQUENCE

seqTyMeaning asn1Envs tyNm ty mp synTags ctls =
  case ctls of
    SimpleComponents comTys ->
      checkedMaybe (distinctEls $ map comTyNm comTys)
      (do
        compEnvs <- getComponentEnvs asn1Envs mp comTys
        Just $ mkSequenceCoders asn1Envs mp tyNm ty synTags
          compEnvs)
    ...

getComponentEnvs :: ASN1_Envs -> ModuleParameters ->
  [ComponentType] -> Maybe [ComponentEnv]

mkSequenceCoders :: ASN1_Envs -> ModuleParameters ->
  IdentType -> Type -> [SyntacticTag] -> [ComponentEnv] ->
  TypeEnv
moduleMeaning :: ASN1_Envs -> ModuleDefinition -> Maybe ASN1_Envs
moduleMeaning asn1Envs md = moduleBodyMeaning asn1Envs mb mp
    where
        mb = moduleBody md
        mp = moduleParmsFromModule md

Input environments are global; result is for this module only
Solving for environments

- The global environments input includes the per-module environments
  - For a single module, the input and output is the same environment pair

\[
\text{moduleMeaning} :: \\
\text{ASN1\_Envs} \to \text{ModuleDefinition} \to \text{Maybe ASN1\_Envs}
\]

- Haskell's lazy evaluation allows such recursive definitions
Other data in type environments

The encoder/decoder pairs are parameterized over lists of tags

We associate lists of tags for each type:

\[ T1 ::= [1][2][42] \text{ INTEGER} \]
\[ T2 ::= [18] T1 \]

When encoding a \( T2 \) value, there are five tags to deal with

We also store any constraints associated with a type, to check values to be encoded, or the results of decoding
Alternative representations?

- Semantics should be a resource for ASN.1 users and implementers
- For broader dissemination, we could express the semantics as conventional mathematics
- A big job - about 5000 Haskell LOC
- For development, Haskell is type-checked, and it's executable
Abstracting over encodings

• There are several sets of rules for encoding types (BER, DER, PER, XER); plus roll-your-own encodings
• We split the semantics into encoding-independent and encoding-specific layers
• In the encoding-independent layer, we produce abstract encodings, which we call data streams
  • No octets
Example data stream

Given the type assignment

\[ T1 ::= [101] \text{BOOLEAN} \]

here's the encoding of the value TRUE:

\[
\text{DataStream} \left[ \left( \text{SemanticTag} \{ \text{semTagValue} = \text{ContextTag} 101, \text{semTagApp} = \text{TaggedExplicit} \}, \text{SemanticTag} \{ \text{semTagValue} = \text{UniversalTag BOOLEANTag}, \text{semTagApp} = \text{TaggedExplicit} \}, \text{PrimDatum} \{ \text{PrimBool True} \} \right) \right]
\]

This is human-readable, unlike an octet list
A more complicated data stream

Given the type

```
SEQUENCE { foo INTEGER, bar BOOLEAN }
```

the encoding of `{ foo 42, bar TRUE }` yields:

```
DataStream
  [[[SemanticTag {semTagValue = UniversalTag SequenceTag,
     semTagApp = TaggedExplicit}],
    AggregateToken SequenceToken],
  [[[SemanticTag {semTagValue = UniversalTag IntegerTag,
     semTagApp = TaggedExplicit}],
     PrimDatum (PrimInteger 42)],
  [[[SemanticTag {semTagValue = UniversalTag BooleanTag,
     semTagApp = TaggedExplicit}],
     PrimDatum (PrimBool True)]]
```
From abstract to concrete

- **Encodings are a vital part of the semantics of ASN.1**
- An abstract data stream contains all the information we need to produce octets for any encoding (that's the goal, at least)
- **Some information could be lost when going to the concrete level**
  - For example, **IMPLICIT tags overwrite other tags, so we couldn't recapture the original abstract data stream from octets alone**
- **We've implemented a translation between abstract data streams and DER**
  - **We build decoder when encoding, so no information is lost**
Type/value compatibility

- X.680 Annex B contains complicated notions of “identical type definitions” and “value mappings” between types
  - Not clear how to use these concepts, except from examples
  - Are examples exhaustive?
- Semantics uses a more principled notion of type and value compatibility
Type/value compatibility, cont.

\[
\begin{align*}
 &a \quad T1 ::= v \quad \text{-- } v \text{ is some value notation} \\
 &b \quad T2 ::= a \\
 &c \quad T3 ::= b \\
\end{align*}
\]

we assess

- the value/type compatibility of \( v \) and \( T1 \)
- the value/type compatibility of \( v \) and \( T2 \)
- the value/type compatibility of \( v \) and \( T3 \)
- the type/type compatibility of \( T1 \) and \( T2 \)
- the type/type compatibility of \( T2 \) and \( T3 \)
Type/value compatibility, cont.

\[ c \longrightarrow b \longrightarrow a ::= v \]

\[ T3 :> T2 :> T1 \]

where \( :> \) means

“there's at least one instance of the RH type that can be mapped to the LH type”
Even more principled ...

- We're working on a set of inference-rule style type rules
- Effectively the same as the code in the semantics, more elegantly presented
- To be shared between semantics and interpreter implementation
Lacunae

- Check that each type is instantiable, i.e., has at least one finite instance
- Consider:
  \[
  T1 ::= \text{SEQUENCE} \{ x \text{ BOOLEAN}, y \text{ } T1 \}
  \]
  - Only infinite values in this case
  - Uninstantiability can be more subtle
- Algorithm by Rinderknecht could be added to semantics
- We're not checking that values appearing in a constraint contains at least one value denoted by the parent type:
  \[
  \text{INTEGER} \ (15..42) \ (11..14)
  \]
Status

- Coded, reviewed at Galois, outside semantics expert

Tests:
- Manual tests of each data type
- Automatically generated tests, including multiple modules
- Round-trip = encode/decode tests

Review of semantics against X.680, clause-by-clause
- Semantics is annotated with relevant sections of X.680

Using semantics as a reference implementation for interpreter testing
- Tried large number of QuickCheck-generated modules
- Automated test harness
TODO

- Add support for more of ASN.1 to semantics
- Use implementation of type inference rules
- Check for type instantiability