An Algorithm for Validating ASN.1 (X.680) Specifications using Set Constraints

Christian Rinderknecht
Network Architecture Laboratory
Information and Communications University
58-4 Hwaam-dong, Yuseong-gu, Daejeon,
305-732, Republic of Korea
rinderkn@nalab.icu.kr

July 2003

Abstract

Abstract Syntax Notation One (ASN.1) is a standard language for defining data types whose values may be exchanged across a network between two communicating applications, independently from the possible heterogeneity of the peers. ASN.1 has been adopted by a wide range of applications, such as network management, secure email, mobile telephony, voice over IP etc. It offers a very involved subtyping paradigm consisting of constraints upon recursive types, which restrict their sets of values in a set-theoretic manner or in a structural way. Because of this great expressiveness, most ASN.1 compilers are not likely to fully check arbitrary combinations of subtyping constraints. We propose to fully validate the X.680 specifications, i.e., the main part of ASN.1, by means of an algorithm which relies on the set constraints theory. Set constraints are inclusions between expressions interpreted over the domain of sets of trees which may be recursively defined. We define a system of constraints which can model all the specifications, we provide a complete collecting algorithm which extracts such constraints from a given specification, and, finally, we give a solving procedure which relies upon an algorithm of Aiken and Wimmers. As a result, either the constraints have no solutions (and the specification must be rejected), or the value sets can be finitely represented. It is straightforward to determine whether these value sets are empty; if they are empty then the specification is rejected. This article addresses both the network tool implementors and the theorist audience.
Keywords: ASN.1, abstract syntax notation, validation, compilation, set constraints.

Introduction

The wide variety of software and hardware architectures in distributed systems and telecommunications makes it valuable to use a common high-level data notation in protocol specifications. For this reason, the ISO organization and the International Telecommunications Union defined the *Abstract Syntax Notation One* (ASN.1) [1–4] series of standards. ASN.1 is a language of data types allowing the protocol designer to capture numerous networking concepts, such as protocol data units, without worrying about the possible environment and implementation heterogeneity of the peers. The peers must share a set of ASN.1 modules and agree upon a method for encoding values (which are produced at run-time by the communicating applications) into series of bits: the *encoding rules* [5, 6]. An ASN.1 compiler accepts a set of ASN.1 modules and, according to a given set of encoding rules and a peer-specific target programming language, produces a set of data type definitions in that programming language, together with a codec for the values to be exchanged. Then these pieces of source code are compiled and linked separately against the communicating application.

ASN.1 has been adopted for a wide range of applications, such as network management, secure email, mobile telephony, air traffic control, video conferencing over the Internet, electronic commerce, digital certificates, radio paging, as well as emerging technologies like interactive television and financial service systems. ASN.1-based software is used in Microsoft's Internet Explorer and Outlook. It is also found in wireless applications from Nokia, Ericsson and Motorola. ASN.1 is used to implement cryptographic protocols which secure credit card purchases over the Internet. Biometrics, databases, ATM transactions, plane take-offs and landings all rely on ASN.1[1].

There are excellent books [7, 8] for the audience of protocol designers and users, but it is still a challenge to write an ASN.1 compiler. The main reason is that, in order to fulfill its users' numerous needs, the language is extremely expressive (without including functions). As a consequence, some compilers may reject valid specifications or, worse, silently accept invalid ones. Vendors argue that this is hardly a real problem because such complex specifications are rarely found in practice. Nevertheless, we claim that this pragmatic approach can fruitfully be enhanced by a theoretical study which leads to an actual implementation. Semantics, i.e. consistent mapping from the syntactic constructs into some set of logical objects, leads to a greater understanding of ASN.1 and the opportunity for a better product. The aim of this work is to provide such a mapping for X.680 [1], the main part of ASN.1[2], by means of an algorithm that can be implemented and integrated in the front-end analyser of an ASN.1 compiler.

Since the paradigm of ASN.1 data types is 'types as sets of values' [9], the main requirement that arises, at least as far as telecom-

[1] See `http://asn1.elibel.tm.fr/` to learn about more uses.

[2] X.680 does not contain information objects, non-subtyping constraints or parameterization.

munication is concerned, is that *types must contain at least one finite value*. The finiteness condition applies no matter what the encoding rules are, but it arises from the fact that the current standard encoding rules cannot handle infinite values, i.e. recursive values. The existence requirement is the main aspect of the validation of ASN.1 specifications, because it is directly related to the values that can be encoded, independently from the encoding rules. It is also the most difficult, because it cannot be dealt with by syntactic means only, i.e. it requires computations or, more generally, inductions on mathematical objects.

In mainstream programming languages, typechecking (i.e. checking whether a value complies with its declared type) is enough as far as validation is concerned and the sets of values corresponding to types are not considered explicitly. In ASN.1, the great deal lies in an involved notion of subtyping. It consists of constraints upon recursive types, which restrict their sets of values in a set-theoretic manner (e.g. by intersection) or in a structural way (e.g. by requiring the omission of some fields in a record-like construct).

In this article we introduce a set-theoretic interpretation which maps types and subtypes into sets of values, by means of an algorithm, and also allows a constructive decision procedure for the 'at least one finite value' property — thus fully validates the X.680 specifications (except tagging). The algorithm deals with the entire X.680 standard. It brings new insights to obscure areas of the standard (like type compatibility in assignments or recursion), which, while not often used by the protocol designer, are unavoidable for the tool implementor concerned with full conformance. The algorithm is twofold: a collecting algorithm, which extracts some constraints, and a solving procedure. Some of the constraints are set constraints [10, 11], so the solving procedure relies upon Aiken and Wimmers' algorithm [12]. Set constraints are inclusions between expressions interpreted over the domain of sets of trees.

First, in section 1, we briefly introduce ASN.1 (the subtyping constraints are presented step by step in section 6). In section 2, we define a strict subset of X.680 which has fewer ambiguities and syntactic constructs; it also allows a much simpler presentation of the collection algorithm of sections 5 and 6. We give a procedure for rewriting every X.680 specification into core ASN.1. We provide, in section 3, a formal predicate for the 'at least one finite value' property on (unconstrained) types in core ASN.1. This property is a prerequisite for handling subtypes. Next, in section 4, we introduce our constraints. The collecting algorithm is introduced in two steps: first, the collection from types is given in section 5; second, the collection from proper subtypes appears in section 6. We finally explain the resolution process in section 7.

1 Presentation of ASN.1

This section provides a very short overview of ASN.1. For a more detailed introduction, please refer to Dubuisson's book [7]. The subtyping features will be presented in section 6 together with the constraint

collection from subtypes. ASN.1 provides basic types as follows.

- The `BOOLEAN` type has two predefined values `TRUE` and `FALSE`, *e.g.,* `ok BOOLEAN ::= TRUE` defines a value `TRUE` whose name is `ok` and whose type is `BOOLEAN`.
- The `NULL` type only has one value, also noted `NULL`. This type is often used as a placeholder in many real complete specifications to indicate that no additional information is needed, or it is used to test incomplete specifications.
- The `INTEGER` type matches the mathematical set $\mathbb{Z}$, *e.g.,* `zero INTEGER ::= 0`. The syntax also allows some constants to be distinguished: `DayInTheYear ::= INTEGER {first (1), last (365)}` defines the type `DayInTheYear` as being `INTEGER`, and distinguishes two integers named `first` and `last`, whose respective values are `1` and `365`. Then `newYearsEve DayInTheYear ::= last` defines a value `newYearsEve`. The definition is valid because `last` is in the scope of `DayInTheYear`; the name `newYearsEve` is bound to the value `365`.
- The `ENUMERATED` type defines a collection of (constant) names, like `SynchroIndicator ::= ENUMERATED {serial, parallel}` allows the following value definition: `synchro SynchroIndicator ::= serial`. It is possible, though not recommended, to specify the *encoding* of an enumerated value, like `PositiveLogics ::= ENUMERATED {false (0), true (1)}`, but this has no impact on the values themselves.
- The `REAL` type corresponds to the mathematical decimal numbers, defined either with a dotted notation, *e.g.,* `5.7`, or a sequence, *e.g.,* `{mantissa 1, base 10, exponent -3}`.
- The `BIT STRING` type corresponds to strings of bits, *e.g.,* `'1101'B` (binary) or `'0D'H` (hexadecimal). The syntax also allows some bits to be distinguished. Given `T ::= BIT STRING {msb (7), lsb (0)}`, the definition `v T ::= {msb, lsb}` stands for `v T ::= '10000001'B`. It is also possible to restrict the size of the string using a subtyping constraint: `StringOf32Bits ::= BIT STRING (SIZE (32))`.
- The `OCTET STRING` type is similar to the `BIT STRING`, except that the encoded strings must contain a number of bits that is a multiple of eight (and no bit can be distinguished by a name).
- The `OBJECT IDENTIFIER` and `RELATIVE-OID` types are used to reference other ASN.1 modules at an international level, by means of a path in a standard tree. They can also identify a physical object, such as a printer on a network, or a postal package, or an ASN.1 type which is carried in some larger message. They are not considered here.
- For historical reasons there are plenty of string types in ASN.1, like `NumericString`, `IA5String`, `UTF8String`, `GeneralString`

etc. They mainly differ in the alphabet they are built upon[3]. Here, we will not make any difference between these strings, and assume there is only one kind, called `String`.

These basic types can be used to construct other types:

- The `SET` type corresponds to the record-like structures in programming languages, *e.g.,* `PersonInfo ::= SET {age INTEGER, married BOOLEAN}` of which one value may be: `i PersonInfo ::= {married TRUE, age 32}`. Some components may be marked as optional or having a default value, *e.g.,* `Point ::= SET {x REAL DEFAULT 0.0, y REAL DEFAULT 0.0}` allows defining the value `origin Point ::= {}`, which is the same as `origin Point ::= {x 0.0, y 0.0}`. Here is an example from a real protocol:
```
DataAcknowledgementTPDU ::= SET {
  destRef         Reference,
  yr-tu-nr        TPDUnumber,
  checkSum        CheckSum OPTIONAL,
  subSeqNr        SubSequenceNumber DEFAULT 0,
  flowControlCnf FlowCntlConf OPTIONAL}
```
- The `SEQUENCE` type is the same as the `SET` type, except that the component values must be given in the same order as they are declared, *e.g.,*, given `Point ::= SEQUENCE {x REAL, y REAL}`, the value `origin Point ::= {y 0.0, x 0.0}` is rejected.
- The `SET OF` type corresponds to the mathematical notion of sets with repetition: all elements are of the same type, but their number is not known beforehand (unless the set's size is constrained to a given value), and they can be repeated, *e.g.,* `T ::= SET OF INTEGER` allows the value definitions `empty T ::= {}` and `small T ::= {7, 9, 1, 1, 3}`.
- The `SEQUENCE OF` type corresponds to the dynamic arrays or lists of some programming languages. It is similar to the `SET OF` type, except that the elements will be encoded in the specified order. Since the encoding rules are out of the scope of this paper, this difference is not relevant.
- The `CHOICE` type corresponds to a `union` in C, a `case` in Pascal, or a sum type in ML. For instance `T ::= CHOICE {x REAL, y BOOLEAN}` allows the following declarations: `u T ::= x : 0.5`, where the component `x` is chosen to build the value, and `v T ::= y : FALSE` where the component `y` is used. The protocol data units are `CHOICE` types, because they model all the possible queries and responses between two peers. As we show later, a `CHOICE` type may be recursive, like the other constructed types. An example from a network management protocol is
```
CMISFilter ::= CHOICE {
```

[3]There are other factors besides just the alphabets. Some string types such as `GeneralString` allow escape characters to kick into alternate character sets (such as those for different languages) while others such as `UTF8String` can represent characters of all languages directly.

```
item FilterItem,
and SET OF CMISFilter,
or SET OF CMISFilter,
not CMISFilter}
```

2 Core ASN.1

It is difficult to separate the different concepts throughout the syntax. The *types*, *values* and *subtyping constraints* may depend on each other: a type may contain constraints (on components) and values (*e.g.,* default values), a value has a declared type, and constraints rely upon types (*e.g.,* inclusion constraint) and values (*e.g.,* value constraint). Another related difficulty is the large number of syntactic constructs.

In order to allow a clearer presentation of the constraint collection (sections 5, 6 and 7), we define a strict subset of X.680, which we call from now on core ASN.1 (versus *full* ASN.1), that will be used in the rest of this paper. In core ASN.1,

- there are no `COMPONENTS OF` or selection types;
- the `INTEGER` type does not allow defining constants;
- component types are references;
- `SET OF` and `SEQUENCE OF` apply to references;
- default values are references;
- enumerated and bit string constants are references;
- types of declared values are references;
- default, enumerated, integer and bit string values appear in a constraint upon their expected type;
- types in inclusion constraints are references;
- there is no type reference just after the symbol '`::=`' and constraints appear only at top-level, *i.e.,* the extended Backus-Naur Form for type declarations is: <type reference> ::= <non-reference type without inner constraints> ["("<subtyping constraint>")"]
- there are no infinite values, *i.e.,* recursive values.

Since we have not yet introduced the collection, it is awkward to explain here the rationale behind core ASN.1. As a consequence, this information will be given later and the reader may skip the next section when reading this for the first time.

2.1 Mapping full ASN.1 into core ASN.1

Full ASN.1 is mapped into core ASN.1 by applying a series of rewritings. It is important to note that each step strictly preserves the expressiveness of full ASN.1. In other words: core ASN.1 can express all that can be expressed in full ASN.1 and nothing more.

Another useful property is that each simplification output can be given in (the syntax of) full ASN.1, making presentation easier. As

software tools use a specific internal data representation, the practical bonus is that pretty-printing is then possible at each stage with the *same* initial pretty-printer (*i.e.,* for full ASN.1).

It is assumed that the following transformations and checkings apply to an ASN.1 module whose syntax complies with X.680 [1]. (The attentive reader will note that not all the rewritings commute, *i.e.,* the following enumeration cannot be arbitrarily shuffled.)

1. We extract the default constant values from the `SEQUENCE` and `SET` types, following the example

   ```
   T ::= SET {a REAL DEFAULT 0.0}
   ⟶ { T ::= SET {a A DEFAULT v}
     { A ::= REAL
     { v A ::= 0.0
   ```

   where `A` is a fresh type reference and `v` is a fresh value reference.

2. We lift the enumeration constants (enumerated, integer and bit string constants) to the top-level, as shown by (`v` is a fresh value reference):

   ```
   T ::= ENUMERATED {a(x), b}
   ⟶ { T ::= ENUMERATED {a(v), b}
     { v INTEGER ::= x
   T ::= INTEGER {a(x)}
   ⟶ { T ::= INTEGER {a(v)}
     { v INTEGER ::= x
   T ::= BIT STRING {a(x)}
   ⟶ { T ::= BIT STRING {a(v)}
     { v INTEGER (0..MAX) ::= x
   ```

3. For each value declaration, we extract the given type and create a corresponding type declaration for it. We also create another type declaration where the previous type is required to contain the originally declared value. For instance, consider

   ```
   y REAL(0..9) ::= 1 ⟶ { y A ::= 1
                        { A ::= REAL(0..9)
                        { B ::= A (y)
   ```

   where `A` and `B` are fresh type references. This way the declared type in a value definition is bound in a type definition (`A`). Also, the typechecking of a value (`y`) can be done with our algorithm (through `B`), since it deals with subtyping constraints.

4. The types which appear in `COMPONENTS OF` constructions are replaced by fresh type references (in the following, `A` is a fresh type reference)

   ```
   T ::= SET {COMPONENTS OF SET {a REAL}}
   ⟶ { T ::= SET {COMPONENTS OF A}
     { A ::= SET {a REAL}
   ```

5. We want to relax the dependence between subtyping constraints and types. Hence, for each inclusion constraint, we replace the included type by a fresh reference and add a corresponding new type declaration, like (`A` is a fresh type reference):

   ```
   T ::= U(SET{a REAL}) → { T ::= U (A)
                          { A ::= SET {a REAL}
   ```

6. We replace each component type by a reference:
   `T ::= SET {a REAL, b SET {d INTEGER}, c U (V)}`
   $\rightarrow \begin{cases} \texttt{T ::= SET \{a A, b B, c C\}} \\ \texttt{A ::= REAL B ::= SET\{d D\} C ::= U(V)} \\ \texttt{D ::= INTEGER} \end{cases}$
   where `A`, `B`, `C` and `D` are fresh type references.
7. At this step, we replace each type to which a `SET OF` or a `SEQUENCE OF` applies, by a reference:
   $\texttt{T ::= SET OF REAL (C)} \longrightarrow \begin{cases} \texttt{T ::= SET OF A} \\ \texttt{A ::= REAL (C)} \end{cases}$
   where `A` is a fresh type reference.
8. We remove the selection types (at top-level)
   $\begin{cases} \texttt{A ::= i < B} \\ \texttt{B ::= C} \\ \texttt{C ::= CHOICE\{i D\}} \\ \texttt{D ::= INTEGER} \end{cases} \longrightarrow \begin{cases} \texttt{A ::= INTEGER} \\ \texttt{B ::= C} \\ \texttt{C ::= CHOICE\{i D\}} \\ \texttt{D ::= INTEGER} \end{cases}$
   You must also be aware of the possibly misleading case:
   $\begin{cases} \texttt{A ::= SET OF S} \\ \texttt{B ::= A (SIZE (7))} \end{cases} \longrightarrow \begin{cases} \texttt{A ::= SET OF S} \\ \texttt{B ::= SET (SIZE (7)) OF S} \end{cases}$
   The result `B ::= SET OF S (SIZE (7))` would be wrong!
   This step is difficult because it removes all recursive types declarations that do not lead to a uniquely defined type, like `T ::= T` or `T ::= CHOICE {a a < T}` etc.
   Note that the selection types that do not define a unique type lead to recursive type definitions whose pattern is `X ::= X`, as
   $\texttt{T ::= CHOICE\{a a < T\}} \longrightarrow$
   $\begin{cases} \texttt{T ::= CHOICE \{a A\}} \\ \texttt{A ::= a < T} \end{cases} \longrightarrow \begin{cases} \texttt{T ::= CHOICE \{a A\}} \\ \texttt{A ::= A} \end{cases}$
   From now on we know exactly what a referenced type is, and thus what is the type of a value.
9. The top-level type references are unfolded, *i.e.,* the type references at the declaration level are replaced by the type they reference
   $\begin{cases} \texttt{T ::= U (C)} \\ \texttt{U ::= REAL (D)} \end{cases} \longrightarrow \begin{cases} \texttt{T ::= REAL (D ^ C)} \\ \texttt{U ::= REAL (D)} \end{cases}$
   During this step, ill-formed recursive definitions, like `X ::= X`, are rejected.
10. The default values are expanded, like
    $\{$ `v T ::= {}  T ::= SET {a U DEFAULT w}`
    $\longrightarrow$ `v T ::= {a w}  T ::= SET {a U DEFAULT w}`
11. The type references in the `COMPONENTS OF` clauses are replaced by their corresponding components
    $\begin{cases} \texttt{T ::= SET \{COMPONENTS OF A\}} \\ \texttt{A ::= SET \{a REAL\}} \end{cases}$
    $\longrightarrow \begin{cases} \texttt{T ::= SET \{a REAL\}} \\ \texttt{A ::= SET \{a REAL\}} \end{cases}$
12. Integer and bit string constants are unfolded
    $\begin{cases} \texttt{T ::= INTEGER \{c(x)\}} \\ \texttt{v T ::= c} \end{cases} \longrightarrow \begin{cases} \texttt{T ::= INTEGER} \\ \texttt{v T ::= x} \end{cases}$

In the case of bit string values which are specified by means of a series of bit names, we unfold their associated references and replace the value by an equivalent one without those names

$$\left\{\begin{array}{l}\texttt{T ::= BIT STRING \{msb(x),lsb(y)\}}\\ \texttt{v T ::= \{msb,lsb\}}\end{array}\right. \longrightarrow \left\{\begin{array}{l}\texttt{T ::= BIT STRING}\\ \texttt{v T ::= '10000001'B}\end{array}\right.$$

where $\left\{\begin{array}{l}\texttt{x INTEGER (0..MAX) ::= 7}\\ \texttt{y INTEGER (0..MAX) ::= 0}\end{array}\right.$ (see step 2).

This step may reveal some recursive values

$$\left\{\begin{array}{l}\texttt{T ::= INTEGER \{c(v)\}}\\ \texttt{v T ::= c}\end{array}\right. \longrightarrow \left\{\begin{array}{l}\texttt{T ::= INTEGER}\\ \texttt{v T ::= v}\end{array}\right.$$

13. We disallow the recursive values, like `v T ::= {v}` or

$$\left\{\begin{array}{l}\texttt{v T ::= \{v\}}\quad \texttt{T ::= SET OF T}\\ \text{or}\\ \texttt{v T ::= \{}\}\quad \texttt{T ::= SET \{a T DEFAULT v\}}\end{array}\right.$$

2.2 Validation issues

In core ASN.1, it is possible that

1. types have only infinite values: `T ::= SET {a T}`
2. values are ill-typed: `v T ::= ""` `T ::= REAL`
3. in particular, value references may be ill-typed:
$$\left\{\begin{array}{l}\texttt{a A ::= b}\quad \texttt{A ::= INTEGER}\\ \texttt{b B ::= 1.5}\quad \texttt{B ::= REAL}\end{array}\right.$$
4. constraints are inconsistent: `T ::= REAL (SIZE(7))`
5. subtypes are empty:
`T ::= SET ((SIZE(1)) INTERSECTION (SIZE(2))) OF REAL;`
6. subtypes have no value set: `T ::= A(ALL EXCEPT T)`

These cases can be classified into the different problems: the *finiteness* problem (case 1), the *typechecking* problem (case 2), the *type compatibility* problem (case 3), the *constraint consistence* problem (case 4), the *non-emptiness* problem (case 5) and the *solvability* problem (case 6). The type compatibility problem is a sub-case of the typechecking problem, and constraint consistence together with non-emptiness are subcases of the solvability problem, because we will explicitly construct the values of each (sub)type when the system is solved. Moreover, since we added a new type declaration for each value declaration at rewriting step 3, the solvability of the subtyping constraints will cope with the typechecking problem. So, finiteness and solvability are enough to get a full validation of X.680 specifications and we need, as a starting point, to express formally those concepts.

2.3 Algorithmic meta-language

We shall use as *meta language* for the description of our algorithm a version of the functional language ML: OCaml[4] [13], which is a full-fledged programming language, as well as, historically, a logic meta-language. Therefore our algorithm is close to an actual implementation and is also a formal (operational) model. Readers familiar with ML may skip this section, which gives a crash overview of its syntax and semantics. (This presentation is based on Pidgin ML [14].)

The core language has types and values. Thus, 1 is a value of predefined type *int*, whereas "CL" is a *string*. Pairs of values inhabit the corresponding product type. Therefore, (1, "CL") has type ($int \times string$). Recursive type declarations create new types, whose values are inductively built from the associated constructors. Thus a type modeling a binary tree of integers could be declared as a sum by: **type** *ib_tree* = Node **of** $ib\_tree \times int \times ib\_tree$ | Leaf. Parametric types give rise to polymorphism: if x is of type t and l is of type (t list), we construct the list adding x to l as $x :: l$. The empty list is [], of (polymorphic) type ($'a$ list). Although the language is strongly typed, explicit type specifications are rarely needed from the programmer, since principal types may be inferred mechanically.

The language is functional in the sense that functions are first class objects. Therefore the integer doubling function may be written as **fun** $x \to x + x$, and it has type $int \to int$. It may be associated to the name *double* by declaring: **let** $double = \textbf{fun}\, x \to x + x$. Equivalently we could write: **let** $double\, x = x + x$. Its application to value n is written as $(double\, n)$ or even $double\, n$ when there is no ambiguity. Application associates to the left, and thus $f\, x\, y$ stands for $((f\, x)\, y)$. Recursive functional values are declared with the keyword **rec**. Thus we may define the factorial function as: **let rec** $fact\, n =$ **if** $n \leqslant 0$ **then** 1 **else** $n \times (fact(n-1))$. Functions may be defined by pattern matching. Thus the first projection of pairs could be defined by: **let** $fst = \textbf{fun}\, (x, y) \to x$ or equivalently (since there is only one pattern in this case) by: **let** $fst\, (x, y) = x$. Pattern-matching is also usable in **match** expressions which generalise case analysis, such as: **match** l **with** [] $\to$ **true** | $\_ \to$ **false**, which tests if list l is empty, using underscore as catch-all pattern.

Evaluation is strict, which means that x is evaluated before invoking f in the evaluation of $(f\, x)$. The **let** expressions allow the sequentialization of computations, and the sharing of sub-computations. Thus **let** $x = fact\, 10$ **in** $x + x$ will compute $fact\, 10$ first, and only once.

Exceptions are declared with the type of their parameters, like in: **exception** Failure **of** *string*. An exceptional value may be raised, like in: *raise* (Failure "div 0") and handled by a **try** switching on exception patterns, like: **try** *expression* **with** Failure $s \to \ldots$ Other imperative constructs may be used, such as references, mutable arrays, while loops and I/O commands, but we shall seldom need them. Sequences of instructions are evaluated in left to right regime in bloc expressions such

[4] `http://www.ocaml.org/`

as: **begin** $e_1; ...; e_n$ **end**.

ML is a *modular* language, in the sense that sequences of type, value and exception declarations may be packed in a structural unit called a *module*, amenable to separate treatment. Modules have types themselves, called *signatures*. Parametric modules are called *functors*. The algorithms presented in this paper will only use this modularity structure to access some library functions — the syntax ought to be self-evident.

Despite the focus in this paper is algorithmic, the readers uninterested in computational details may think of ML definitions as recursive equations over inductively defined algebras.

2.4 Abstract grammar

Let us use OCaml's algebraic type declarations to define the *abstract grammar* of core ASN.1. This grammar captures the syntactically correct constructs of core ASN.1, except those which involve *tags* [1, §3.6.69, §8] (since they are related to the encoding rules) and the OBJECT IDENTIFIER and RELATIVE OID types and values (for the sake of brevity). The parser's output is a pair of a type environment and a value environment. The former is a mapping from type names to subtypes, corresponding to the type declarations in the ASN.1 specification, and the latter is a mapping from value names to values, corresponding to the value declarations. The subtypes and values are *abstract syntax trees*, complying with the abstract grammar. We do not follow the syntactic conventions of OCaml exactly, as detailed below.

- in mutual recursive polymorphic variant definitions, we shall allow type names instead of a variant, like **type** t = [‘K] **and** u = [‘L | t] (this limitation can be circumvented by an implementation trick out of scope here);
- We allow ASN.1 symbols or keywords as data constructors, like ‘**<**’, ‘**..**’, MINUS-INFINITY or OCTET STRING; we then use underscores to denote the location of their arguments, as in ‘ __ < __ ’;
- We sometimes write COMPONENTS OF T σ instead of the correct (non-currified) COMPONENTS OF (T, σ), and similarly for other data constructors.

The abstract grammar for core ASN.1 values is defined as follows. Firstly, we assume that the parser removes the ambiguity between enumeration constants [1, §19] and value references [1, §11.4]. For instance, in ‘`a T ::= b`’, the token `b` can denote either an enumeration constant or a value reference, depending on the definition of the type `T`. The ambiguity can always be removed just by looking at the type definition (this is easy in core ASN.1). We start by defining the OCaml type *v_ref* for value references, the type *item*, which is used later in the definition of enumerated constants, and the type *label*, which denotes component names

type *v_ref* = [‘VRef **of** *string*]

type *item* = *string* **and** *label* = *string*

The values of ‘VRef’s argument are noted y. Values of type *item* are noted a, b, c, and lists of such values are noted I. Values of type *label* are noted l, and sets of labels are noted $\mathcal{L}$. Let us now define the numeric types straightforwardly:

type *integer* = [‘PosInt **of** *int* | ‘NegInt **of** *int*]
type *real* = [‘PosReal **of** *float* | ‘NegReal **of** *float*]

where ‘PosInt means ‘Positive or null integer’. Next, the OCaml type for ASN.1 values is named $\mathcal{V}$:

type $\mathcal{V}$ = [‘List **of** $\mathcal{V}$ list | ‘Map **of** *label* $\rightarrow \mathcal{V}$
| ‘Nil | TRUE | FALSE | ‘String **of** *string* | *integer*
| ___ **:** ___ **of** *label* $\times \mathcal{V}$ | ‘Enum **of** *item*
| ‘HexStr **of** *string* | ‘BinStr **of** *string* | NULL | *real*
| PLUS-INFINITY | MINUS-INFINITY | *v_ref*]

where ‘List corresponds to values of SET OF and SEQUENCE OF types [1, §25, §27]; ‘Map models values of the SET and SEQUENCE types [1, §24, §26] (the argument is a mapping from labels to values); ‘Nil captures the ambiguous value {}, which can be of type SET OF, SEQUENCE OF, SET, SEQUENCE or BIT STRING; TRUE and FALSE are the values of the BOOLEAN type; ‘String stands for all kinds of character strings; ___ **:** ___ corresponds to CHOICE values [1, §28] (thus its argument is a pair of a label and a value); ‘Enum models enumerated constants; ‘HexStr corresponds to OCTET STRING values [1, §22]; ‘BinStr represents BIT STRING constants [1, §21]; NULL captures the special NULL value [1, §23]; PLUS-INFINITY and MINUS-INFINITY are special values of type REAL.

OCaml values of type $\mathcal{V}$ will be noted v, and lists of values L. The value environments are noted Δ, and have type *string* $\rightarrow \mathcal{V}$. The argument of the ‘Map data constructor is noted M or $\mathrm{M}_{\mathcal{L}}$ when the set of labels (the domain of the mapping) is $\mathcal{L}$ (the empty mapping is simply noted {}). The OCaml recursive type $\mathcal{T}$ represents the core ASN.1 types:

type $\mathcal{T}$ = [CHOICE **of** *label* $\rightarrow$ *t_ref* | OCTET STRING
| SET **of** *components* | INTEGER | ‘String | NULL
| SEQUENCE OF **of** *t_ref* | REAL | BIT STRING
| SET OF **of** *t_ref* | SEQUENCE **of** *components*
| ENUMERATED **of** *item* list | BOOLEAN | *t_ref*]
and *t_ref* = [‘TRef **of** $\mathcal{R}$] **and** $\mathcal{R}$ = *string*
and *components* =
label $\rightarrow$ *t_ref* $\times$ [OPTIONAL | DEFAULT **of** *v_ref*] option

The type *t_ref* denotes type references. The type $\mathcal{R}$ is the countable set of type reference names. The type *components* defines the components of SET and SEQUENCE core ASN.1 types: it is a function from labels to pairs whose first projection is the type reference (in core ASN.1, component types are references) and whose second projection models the component's optional annotation, which tells whether the component is mandatory, optional or has a default value.

OCaml values of type $\mathcal{T}$ are noted T. Values of type $\mathcal{R}$ are noted x. The mapping of type *label* $\rightarrow$ *t_ref*, which is the argument of CHOICE, is noted F, or $\mathrm{F}_{\mathcal{L}}$ when the labels range over $\mathcal{L}$ (in general, the domain of a mapping can be put in subscript), *e.g.,* CHOICE $\mathrm{F}_{\mathcal{L}}$. Values of type *components* are mappings noted Φ, *e.g.,* SET Φ.

ASN.1 subtyping constraints [1, §45] are modelled by the following type $\mathcal{C}$, whose values are noted C:

type $\mathcal{C}$ = [___ UNION ___ **of** $\mathcal{C} \times \mathcal{C}$ | *interval*
| ___ INTERSECTION ___ **of** $\mathcal{C} \times \mathcal{C}$ | $\mathcal{V}$
| ___ EXCEPT ___ **of** $\mathcal{C} \times \mathcal{C}$ | ALL EXCEPT **of** $\mathcal{C}$
| FROM **of** $\mathcal{C}$ | SIZE **of** $\mathcal{C}$ | INCLUDES **of** *t_ref*
| WITH COMPONENTS **of**
kind $\times$ (*label* $\rightarrow$ $\mathcal{C}$ option $\times$ *status* option)
| WITH COMPONENT **of** $\mathcal{C}$ | PATTERN **of** *string*]
and *kind* = Partial | Full
and *status* = [PRESENT | ABSENT | OPTIONAL]
and *interval* = ___ .. ___ **of**
bound $\times$ *in_out* $\times$ *in_out* $\times$ *bound*
and *bound* = [MIN | MAX | 'String **of** *string* | *real*
| 'HexStr **of** *string* | 'BinStr **of** *string* | *integer*]
and *in_out* = < | $\leqslant$

The OCaml type *subtype* models ASN.1 subtypes. An ASN.1 subtype is a pair of a type and an optional subtyping constraint. OCaml values of type $\mathcal{C}$ option are noted σ. The type environments are noted Γ, and have type *string* $\rightarrow$ *subtype*.

type *subtype* = $\mathcal{T} \times (\mathcal{C}$ option)

3 Well-founded Types

In this section we tackle the finiteness problem of ASN.1 types. The solution shall be used later for solving the same problem over subtypes. At this point, let us notice again that *some types have only infinite values*. For instance, `T ::= SET {a T}` is forbidden by the standard since the current encoding rules cannot handle values of such a type (the encoding engine generated by the ASN.1 compiler would loop forever). Let us call *well-founded* a core ASN.1 type which has at least one finite value. Let Γ be a type environment. Let us write $\Gamma \Vdash \mathrm{T}$ if and only if T is well-founded in Γ.[5] For technical reasons, we need another definition: $\Gamma, \mathrm{H} \Vdash \mathrm{T}$ where H (read 'history') is a

[5]It is possible to give a formal and direct definition of this concept. Here is a sketch. First, we define a function $h_\Gamma : \mathcal{V} \rightarrow \mathbb{N} \cup \{+\infty\}$, which computes the maximum height of the abstract syntax tree corresponding to a value in the environment Γ, modulo references: an empty tree, a leaf and a non-cyclic reference add no height (the reference is unfolded then); a node adds a height of 1, and a cyclic reference returns $+\infty$. Second, we define a predicate $\Gamma \vdash v : \mathrm{T}$, read: '$v$ is of type T in the environment Γ'. Then T is well-founded if, and only if, $\exists v \in \mathcal{V}$ such as $h_\Gamma(v) \in \mathbb{N}$ and $\Gamma \vdash v : \mathrm{T}$. We should then prove that our forthcoming axiomatisation of $\Gamma \Vdash \mathrm{T}$ is equivalent to $\exists v \in \mathcal{V}$ such as $h_\Gamma(v) \in \mathbb{N}$ and $\Gamma \vdash v : \mathrm{T}$.

set of type reference names. These names correspond to the previously encountered type references: they allow some recursions to be detected and rejected. By definition: $\Gamma \Vdash \mathrm{T}$ is equivalent to $\Gamma, \varnothing \Vdash \mathrm{T}$. This latter relationship is the smallest one induced by the closure of the following inference rules. Please note that we use '$x \lhd y$' as a shorthand for '**match** x **with** $y \to$ **true**' in OCaml (projection), and **as** has the same meaning as in OCaml (pattern binder):

$$\frac{\neg(\mathrm{T} \lhd \text{`}\mathsf{TRef}\ \_\_ \mid \textsc{Choice}\ \_\_) \quad \neg(\mathrm{T} \lhd \textsc{Set}\ \_\_ \mid \textsc{Sequence}\ \_\_)}{\Gamma, \mathrm{H} \Vdash \mathrm{T}}\ \textsc{Axioms}$$

The rule Axioms states that the types that differ from 'TRef, Choice, Set and Sequence are always well-founded, e.g. `T ::= SET OF T` is well-founded.

$$\frac{x \notin \mathrm{H} \qquad \Gamma(x) \lhd (\mathrm{T}, \sigma) \qquad \Gamma, \{x\} \cup \mathrm{H} \Vdash \mathrm{T}}{\Gamma, \mathrm{H} \Vdash \text{`}\mathsf{TRef}\,(x)}\ \textsc{Ref}$$

The rule Ref handles the case of the type references. The first premise is a look-up in the history to check for a previous occurrence of the reference name: if present, the type is rejected, like `T ::= CHOICE {a T}`. The second premise is a look-up in the specification for the definition of the referenced type. The last premise is the checking of the referenced type.

$$\frac{\Gamma, \mathrm{H} \Vdash \mathrm{F}(l)}{\Gamma, \mathrm{H} \Vdash \textsc{Choice}\,\mathrm{F}_{\{l\} \sqcup \mathcal{L}}}\ \textsc{Choice}$$

In the rule Choice, we use the the symbol $\sqcup$ for the disjoint set union. This rule handles the case of Choice types. The first premise is the projection of a component, which is checked in the following premise: a Choice is well-founded if and only if one of its component is well-founded, e.g. `T ::= CHOICE {a T, b INTEGER}`.

$$\frac{\Gamma, \mathrm{H} \Vdash \textsc{Set}\ \Phi}{\Gamma, \mathrm{H} \Vdash \textsc{Sequence}\ \Phi}\ \textsc{Seq}$$

The rule Seq simply states that the proof of well-foundedness of a Sequence type is the same as the one for the Set with the same components.

$$\Gamma, \mathrm{H} \Vdash \textsc{Set}\ \{\}\ \varnothing\text{-}\textsc{Set}$$

The axiom ∅-Set says that an empty Set type is well-founded.

$$\frac{\Phi(l) \lhd (\mathrm{T}', \mathsf{Some}\,\textsc{Optional}) \qquad \Gamma, \mathrm{H} \Vdash \textsc{Set}\ \Phi_{\mathcal{L}}}{\Gamma, \mathrm{H} \Vdash \textsc{Set}\ \Phi_{\{l\} \sqcup \mathcal{L}}}\ \textsc{SetOpt}$$

The rule SetOpt applies when a component is marked as Optional. In this case, it is ignored, and the remaining components are checked.

Indeed, an optional component can be absent in the value definition, thus any recursion throughout it is valid.

$$\frac{\Phi(l) \lhd (\mathrm{T}', \mathsf{None} \mid \mathsf{Some}\,(\textsc{Default}\ \_\_)) \qquad \Gamma, \mathrm{H} \Vdash \mathrm{T}' \qquad \Gamma, \mathrm{H} \Vdash \textsc{Set}\ \Phi_{\mathcal{L}}}{\Gamma, \mathrm{H} \Vdash \textsc{Set}\ \Phi_{\{l\} \sqcup \mathcal{L}}}\ \textsc{SetDef}$$

The rule SetDef applies when a component is not marked as Optional (first premise): then the type of this component is checked. This allows to reject for instance `T ::= SET {a T}`. The last premise corresponds to the checking of the remaining components.

It is not too difficult to see that if a type satisfies our formal criterion, then it has at least one finite value (the proof tree is isomorphic to value construction steps, i.e. each judgement $\Gamma \Vdash \mathrm{T}$ can be associated to a value of T). Also, our inference system can be considered as an algorithm: just consider the rules and the premises ordered as they are written. Also, the implicit existential quantifiers (on l in rules Choice, SetOpt and SetDef) are easy to make constructive: simply try the components in the given order.

It is worth remarking that the well-foundedness of a type does not imply that its subtypes have at least a finite value. For example, `T ::= CHOICE {a T, b REAL}` is well-founded, but `U ::= T (WITH COMPONENTS {..., b ABSENT})` has no finite value. That is why the relationship $\Vdash$ will be reused in the forthcoming constraint-collecting algorithm, which assumes that all the types in core ASN.1 are well-founded.

4 Constraints

Till now, we have considered all the problems except solvability of subtyping constraints (see section 2.2). Let us start by quoting Olivier Dubuisson [7, §13.11]:

> '[...] the set operators potentially apply to any subtype constraints; the problem lies in the interpretation of the constraints in terms of sets of values to actually determine the possible values of the resulting type. The designers should be aware, however, that the ASN.1 compilers are not likely to check for [...] eccentric constraint combinations.'

The aim of this section is to provide a formal definition of the constraints, which will allow us (in section 7) to determine the values of each subtype in a specification. The integration of this procedure into the analysis phase of an ASN.1 compiler frees the protocol designer from being 'eccentric' or not. The idea is to collect constraints from the specification, and then solve them. Most of our constraints are or rely upon *set constraints*. In the 1990s, the field of the set constraints has been extensively explored, and both major theoretical results (such as complexity for several classes of constraints, or links to other fields such as the tree automata theory) and practical achievements (such as

constraint programming languages or static programme analysis) have been brought to the light.

On one hand, the ASN.1 values have a tree structure, thus fit perfectly the usual domain of set constraints. On the other hand, the ASN.1 subtyping constraints are too general to be modeled only with set constraints: there are also constraints expressed in terms of intervals, regular expressions and powersets. (Our specific contribution is the use and resolution of these special constraints.) So, in order to unify the representation of these concepts, we need to abstract the values and make them simple constraints. We could directly reuse the abstract grammar (type $\mathcal{V}$ in section 2.4), nevertheless it is worth abstracting further the values at this stage; for instance, from a semantic point of view, it is more suitable to consider that bit strings, octet strings, and general strings are all described by regular expressions. Also, it is more uniform to consider that an integer is in fact an interval reduced to one element. In order to simplify the introduction of the collection algorithm, we gather in the same definition the abstracted values, intervals, regular expressions, sets and powersets (they will be separated before the solving procedure, because they require specific algorithms):

Definition 4.1 (Expressions)**.** *An* expression *e is an element of the set* E *defined using the following grammar and OCaml type definitions:*

$$\begin{aligned}
\mathrm{E} ::= {} & \dot{1} \mid \dot{0} \mid \alpha \mid \dot{\neg}\mathrm{E} \mid \mathrm{E}_0 \mathbin{\dot{\cup}} \mathrm{E}_1 \mid \mathrm{E}_0 \mathbin{\dot{\cap}} \mathrm{E}_1 \mid \mathrm{E}_0 \mathbin{\dot{\setminus}} \mathrm{E}_1 \\
& \mid \mathit{series} \mid \mathit{map} \mid __ : __ \ \textbf{of}\ \mathit{label} \times \mathrm{E} \mid \textsc{Null} \\
& \mid \mathit{real_interval} \mid \mathit{closed_int_interval} \mid \textsc{True} \\
& \mid \textsc{False} \mid \text{`Enum}\ \textbf{of}\ \mathit{item} \mid \text{`Regexp}\ \textbf{of}\ \mathit{string} \\
& \mid \mathrm{E}\diamond \mathit{closed_int_interval}
\end{aligned}$$

$$\begin{aligned}
&\textbf{and}\ \mathit{series} = [\text{`Cons}\ \textbf{of}\ \mathrm{E} \times \mathit{series} \mid \text{`Nil}] \\
&\textbf{and}\ \mathit{map} = [\text{`Bind}\ \textbf{of}\ \mathit{identifier} \times \mathrm{E} \times \mathit{map} \mid \text{`Nil}] \\
&\textbf{and}\ \mathit{real_interval} = [\ __\ __\ ..\ __\ __\ \textbf{of} \\
&\quad \mathit{real_bound} \times \mathit{in_out} \times \mathit{in_out} \times \mathit{real_bound}] \\
&\textbf{and}\ \mathit{real_bound} = [\mathit{real} \mid \text{`MinInfReal} \mid \text{`PlusInfReal}] \\
&\textbf{and}\ \mathit{closed_int_interval} = \\
&\quad [\text{`Interval}\ \textbf{of}\ \mathit{int_bound} \times \mathit{int_bound}] \\
&\textbf{and}\ \mathit{int_bound} = [\mathit{integer} \mid \text{`MinInfInt} \mid \text{`PlusInfInt}]
\end{aligned}$$

As a special case, we define some useful constants:

$$\begin{aligned}
&\textbf{let}\ \ \mathbb{N} = \text{`Interval}\,(\text{`MinInfInt}, \text{`PlusInfInt}) \\
&\textbf{and}\ \mathbb{N}^+ = \text{`Interval}\,(\text{`PosInt}\,(0), \text{`PlusInfInt}) \\
&\textbf{and}\ \mathbb{R} = \text{`MinInfReal} < .. < \text{`PlusInfReal}
\end{aligned}$$

The constant $\dot{1}$ *denotes the set of all terms (see definition 7.3);* $\dot{0}$ *the empty set;* α *is a variable denoting an expression (see definition 7.2);* $\dot{\neg}$ *is the complement operator;* $\dot{\cup}$, $\dot{\cap}$ *and* $\dot{\setminus}$ *are straightforward. These operators are the same as Aiken and Wimmers' ones (except their notation is not dotted). Next is the* series *type, which captures a list of expressions; it will be used for encoding sets of ASN.1 values of type* Set of *and* Sequence of*. Next comes the* map *type, which represents mappings from identifiers to expressions; it will be used to denote*

sets of ASN.1 values of type SET *and* SEQUENCE. *Next, there is the* $\_\_ : \_\_$ *constructor that associates a label with an expression; it will be used for encoding values of the ASN.1 type* CHOICE. *The* NULL *constructor is for the* NULL *value. Next, we find the* real_interval *type that defines an interval on the* REAL *values. Next, we get the* closed_int_interval *type which represents the closed intervals on the* INTEGER *values (in theory, it is not necessary to have closed intervals, but this design decision makes the algorithm simpler). The constructors* TRUE *and* FALSE *are obvious. Following, we have the* ‘Enum *constructor which denotes the enumerated constants; then* ‘Regexp *which denotes the ASN.1 regular expressions (we call* regexp *for short); then the integer type; then the real type. The constructor* $\diamond$ *denotes a powerset (see below). The constructors* ‘MinInfReal, ‘PlusInfReal *stand respectively for* $-\infty$ *and* $+\infty$ *on the ASN.1 real numbers;* ‘MinInfInt, ‘PlusInfInt *are* $-\infty$ *and* $+\infty$ *on the integers.*

The mapping from abstracted values to expressions is quite straightforward. Just notice that integer and real numbers are mapped into intervals, and that the value reference names (of type *string*) are cast into the set V of variables:

Definition 4.2 (From values to expressions)**.** **let rec** $\mu : \mathcal{V} \to \mathrm{E} =$ **function**

$$
\begin{array}{l}
\quad \text{`List}\,[\,] \mid \text{`Map}\,\{\} \to \text{`Nil} \\
\mid \text{`List}\,(v :: \mathrm{L}) \to \text{`Cons}\,(\mu\,(v), \mu\,(\text{`List}\,(\mathrm{L}))) \\
\mid \text{`Map}\,(\mathrm{M}_{\{l\} \sqcup \mathcal{L}}) \to \text{`Bind}\,(l, \mu\,(\mathrm{M}(l)), \mu\,(\text{`Map}\,(\mathrm{M}_{\mathcal{L}}))) \\
\mid \text{`BinStr}\,(s) \mid \text{`HexStr}\,(s) \mid \text{`String}\,(s) \to \text{`Regexp}\,(s) \\
\mid (\text{`PosInt}\ \underline{\quad} \mid \text{`NegInt}\ \underline{\quad})\ \mathbf{as}\ v \to \text{`Interval}\,(v, v) \\
\mid (\text{`PosReal}\ \underline{\quad} \mid \text{`NegReal}\ \underline{\quad})\ \mathbf{as}\ v \to v \leqslant .. \leqslant v \\
\mid \textsc{Minus-infinity} \to \text{`MinInfReal} \leqslant .. \leqslant \text{`MinInfReal} \\
\mid \textsc{Plus-infinity} \to \text{`PlusInfReal} \leqslant .. \leqslant \text{`PlusInfReal} \\
\mid l : v \to l : \mu(v) \mid \text{`VRef}\,(y) \to (y : V) \\
\mid v \to v
\end{array}
$$

The next step is to build the constraints on top of the expressions:

Definition 4.3 (Constraints)**.** *A* constraint κ *is a conjunction of inclusions over expressions. The set of constraints is:* $\mathcal{K} ::= \mathcal{K}_0 \ddot{\wedge} \mathcal{K}_1 \mid \mathrm{E}_0 \ddot{\subseteq} \mathrm{E}_1 \mid \mathrm{E}_0 \ddot{=} \mathrm{E}_1$.

Note that the notations for the operators defining the constraints are double-dotted, e.g. $\ddot{\wedge}$. The operator $\ddot{=}$ stands for the double inclusion.

Let us consider the sets of value sets associated with SET OF and SEQUENCE OF types, i.e. powersets of values. For instance, `A ::= SET (SIZE (4..7)) OF INTEGER` denotes the set of integer sets whose cardinals belong to the interval $[4; 7]$. For each value of the cardinal it is possible to give the corresponding integer set expression, and then make the union to get the expression associated with type `A`. In general, this encoding is bulky, it makes the constraint solving inefficient and it is unable to cope with the infinite bound $\mathbb{N}^+$. A better idea is to keep the interval of cardinals *together with* an over-approximation of

the powerset itself (this latter contains sets of any size). The powerset expression $e\diamond\varsigma$ is a pair of an expression e which denotes a powerset (coming from SET OF and SEQUENCE OF types), and an interval ς of the elements' cardinals (coming from SIZE subtyping constraints).

Our idea is to analyse the subtypes in core ASN.1 and to produce a mixed constraint for each one. Since component types are all type references (see section 2.1, step 6) and type declarations are of the form <type reference> ::= <non-reference type without inner constraints> (<subtyping constraint>) or simply <type reference> ::= <non-reference type without inner constraints> (see step 9), we can parse each subtyping constraint without destructuring the type it applies to (indeed, the constraints on component types are only found at the top-level). Thanks to this specific shape of core ASN.1, the collection process has two weakly interdependent aspects: the collection on types and the collection on (proper) subtypes. The link between the two collections is due to the component types, i.e. type references, because a type reference can denote either a type or a proper subtype; hence this link will appear in the treatment of the 'TRef constructor.

5 Constraints from Types

In this section we define the collection of constraints from types (i.e. declarations of pattern <type reference> ::= <non-reference type without inner constraints>). It is a mapping $\mathcal{T} \to V \to (\mathcal{R} \to V) \to \mathcal{K}$, where $\mathcal{T}$ is the set of types, V is the set of variables (ranged over by α, β, γ etc.), $\mathcal{R}$ is the countable set of type reference names (ranged over by x) and $\mathcal{K}$ is the set of constraints. The notation for the mapping is $[\![\mathrm{T}]\!]_\alpha(\mathcal{Q})$, where T is the analysed type, α is the variable denoting the terms corresponding to the values of T and $\mathcal{Q}$ is a mapping from type reference names into the variables, which allows the proper handling of the recursive types. For the sake of clarity, we write sometimes $\mathcal{Q}_\mathcal{R}$ to show that the domain of $\mathcal{Q}$ is $\mathcal{R}$.

For instance: $[\![\textsc{Null}]\!]_\alpha(\mathcal{Q}) = (\alpha \doteqdot \textsc{Null})$ means that the set constraint associated to the NULL type, whose set of terms is denoted by α, is $\alpha \doteqdot \textsc{Null}$; that is to say the terms are exactly the NULL set constant. The solution of this constraint is the substitution that maps the variable α to the set $\{\textsc{Null}\}$. Let us consider the constraint collection from the CHOICE type:

$$[\![\textsc{Choice}\ \mathrm{F}_{\{l\}}]\!]_\alpha(\mathcal{Q}) = \begin{array}{l}\textbf{let}\ \beta\ \textit{be a fresh variable}\\ \textbf{in}\ [\![\mathrm{F}(l)]\!]_\beta(\mathcal{Q})\ \ddot{\wedge}\ \alpha \doteqdot l : \beta\end{array}$$

This equation is for CHOICE types having only one component, labeled l. The expression corresponding to the component $\mathrm{F}(l)$ is denoted by the variable β which is used to build the constraint for the CHOICE type.

$$\begin{array}{l}[\![\textsc{Choice}\ \mathrm{F}_{\{l,m\}\sqcup\mathcal{L}}]\!]_\alpha(\mathcal{Q}) =\\ \quad\textbf{let}\ \beta\ \textit{and}\ \gamma\ \textit{be fresh variables}\\ \quad\textbf{in}\ \ [\![\textsc{Choice}\ \mathrm{F}_{\{m\}\sqcup\mathcal{L}}]\!]_\gamma(\mathcal{Q})\ \ddot{\wedge}\ [\![\mathrm{F}(l)]\!]_\beta(\mathcal{Q})\\ \qquad\ddot{\wedge}\ \alpha \doteqdot (l : \beta)\,\dot{\cup}\,\gamma\end{array}$$

This equation is for CHOICE types having at least two components, l and m. First, we compute the set constraint for the CHOICE with the same components except l (the expression is named γ). Then there is the constraint for the component l (variable β), and finally the constraint on the whole CHOICE type, whose expression is α and is the union of the expression of the component l and γ.

$$
\begin{array}{l}
[\![\textsc{Set of } \mathrm{T}_0]\!]_\alpha(\mathcal{Q}) = \\
\quad \textbf{let } \beta \textit{ and } \gamma \textit{ be fresh variables} \\
\quad \textbf{in } \; [\![\mathrm{T}_0]\!]_\beta(\mathcal{Q}) \ddot{\wedge} \ \gamma \doteqdot \text{`}\mathsf{Cons}\,(\beta,\gamma) \dot{\cup} \text{`}\mathsf{Nil} \ddot{\wedge} \ \alpha \doteqdot \gamma \diamond \mathbb{N}^+
\end{array}
$$

The variable β denotes the expression from type T, defined by the constraint $[\![\mathrm{T}]\!]_\beta(\mathcal{Q})$. The constraint $\gamma \doteqdot \text{`}\mathsf{Cons}\,(\beta,\gamma) \dot{\cup} \text{`}\mathsf{Nil}$ defines the powerset γ over β sets. These sets are always finite because the types in core ASN.1 are well-founded (see section 3). Finally, the powerset constraint $\alpha \doteqdot \gamma \diamond \mathbb{N}^+$ states that the powerset expression we are looking for is the pair of γ (the powerset itself) and $\mathbb{N}^+$ (the allowed cardinals of the elements of γ). Note that, because we are in core ASN.1, the SET OF applies to a type (T) which is always a reference without subtyping constraint (see section 2.1, step 7).

$$[\![\textsc{Sequence of } \mathrm{T}]\!]_\alpha(\mathcal{Q}) = [\![\textsc{Set of } \mathrm{T}]\!]_\alpha(\mathcal{Q})$$

This equation defines the set of terms of the SEQUENCE OF type as being the same as the SET OF terms. Indeed, the difference between these two types is only meaningful for the encoding rules (hence, at the application level).

$$[\![\textsc{Sequence } \{\}]\!]_\alpha(\mathcal{Q}) = \alpha \doteqdot \text{`}\mathsf{Nil}$$

This equation is for empty SEQUENCE types.

$$
\begin{array}{l}
[\![\textsc{Sequence } \Phi_{\{l\} \sqcup \mathcal{L}}]\!]_\alpha(\mathcal{Q}) = \\
\quad \textbf{let } \beta \textit{ and } \gamma \textit{ be fresh variables} \\
\quad \textbf{in } \; [\![\textsc{Sequence } \Phi_{\mathcal{L}}]\!]_\gamma(\mathcal{Q}) \\
\qquad \ddot{\wedge} \ \textbf{match } \Phi(l) \textbf{ with} \\
\qquad\quad (\mathrm{T}, \mathsf{None} \mid \mathsf{Some}\,(\textsc{Default } \_\_)) \rightarrow \\
\qquad\qquad [\![\mathrm{T}]\!]_\beta(\mathcal{Q}) \ \ddot{\wedge} \ \alpha \doteqdot \text{`}\mathsf{Bind}\,(l, \overline{\beta,\gamma}) \\
\qquad\quad \mid (\mathrm{T}, \mathsf{Some}\,\textsc{Optional}) \rightarrow \\
\qquad\qquad [\![\mathrm{T}]\!]_\beta(\mathcal{Q}) \ \ddot{\wedge} \ \alpha \doteqdot \text{`}\mathsf{Bind}\,(l,\beta,\gamma) \dot{\cup} \gamma
\end{array}
$$

This equation assumes that the set has at least one component, labeled l. We need first to extract the constraint from the SEQUENCE without this component: $[\![\textsc{Sequence } \Phi_{\mathcal{L}}]\!]_\gamma(\mathcal{Q})$. Next, a case analysis is done on the component, $\Phi(l)$. In all cases, the variable β denote the set of terms of the component type: $[\![\mathrm{T}]\!]_\beta(\mathcal{Q})$. Hence we add the constraint $\alpha \doteqdot \text{`}\mathsf{Bind}\,(l,\beta,\gamma)$, for it represents the set of terms when the component l is always present. That is why, when this component can be omitted (see the OPTIONAL case) we complete the set with γ. Note that, because we are in core ASN.1, the (possible) default value is a reference (see section 2.1, step 1), hence we do not care here about it (in particular, we do not constrain it to belong to the set of terms of the component). Indeed, this value, like all initial top-level values,

have been introduced in a so-called *single value* subtyping constraint (see step 3) upon their expected type and which will be considered independently from the current case. Another detail worth mentioning is the lack, in core ASN.1, of COMPONENTS OF clause (see section 2.1, steps 4 and 11).

$$[\![\text{SET}\,\Phi]\!]_\alpha(\mathcal{Q}) = [\![\text{SEQUENCE}\,\Phi]\!]_\alpha(\mathcal{Q})$$

This equation states that the constraint for the SET type is the one for the SEQUENCE with the same components. This is in fact an approximation. Indeed, the difference between SET and SEQUENCE is that the values of the latter must be given in the same order as the components are given. Introducing explicitly the proper combinatorics for the SET values would result in exponential size of the term set. So we approach the SET values as if they were SEQUENCE values. In theory, after the equation solving, we would have to consider these values modulo permutation (except for the validation purpose, which only requires the existence of one term in the set).

$$\begin{aligned}
[\![\text{INTEGER}]\!]_\alpha(\mathcal{Q}) &= \alpha \doteq \mathbb{N} \\
[\![\text{REAL}]\!]_\alpha(\mathcal{Q}) &= \alpha \doteq \mathbb{R} \\
[\![\text{BOOLEAN}]\!]_\alpha(\mathcal{Q}) &= \alpha \doteq \text{TRUE} \,\dot\cup\, \text{FALSE} \\
[\![\text{NULL}]\!]_\alpha(\mathcal{Q}) &= \alpha \doteq \text{NULL} \\
[\![\mathsf{String}]\!]_\alpha(\mathcal{Q}) &= \alpha \doteq \text{`}\mathsf{Regexp}\ \texttt{".*"} \\
[\![\text{BIT STRING}]\!]_\alpha(\mathcal{Q}) &= \alpha \doteq \text{`}\mathsf{Regexp}\ \texttt{"[\textbackslash s01]*"} \\
[\![\text{OCTET STRING}]\!]_\alpha(\mathcal{Q}) &= \alpha \doteq \text{`}\mathsf{Regexp}\ \texttt{"[\textbackslash s\textbackslash da-fA-F]*"}
\end{aligned}$$

$$\begin{aligned}
[\![\text{ENUMERATED}\,[a]]\!]_\alpha(\mathcal{Q}) &= \alpha \doteq \text{`}\mathsf{Enum}\,(a) \\
[\![\text{ENUMERATED}\,(a::b::\text{I})]\!]_\alpha(\mathcal{Q}) &= \\
&\quad \textbf{let}\ \beta\ \textit{be a fresh variable} \\
&\quad \textbf{in}\ [\![\text{ENUMERATED}\,(b::\text{I})]\!]_\beta(\mathcal{Q}) \ \ddot\wedge\ \alpha \doteq \text{`}\mathsf{Enum}\,(a) \,\dot\cup\, \beta
\end{aligned}$$

These equations deal with the constraints from the basic types of ASN.1. You may notice that the terms of the types BIT STRING, OCTET STRING and String are encoded using the regular expressions. Also, we do not care about the enumerated and bit string constants. Indeed, in core ASN.1, these values, like all initially declared values, have been introduced in single value subtyping constraints (see section 2.1, steps 2 and 3) on their expected type, and which wil be considered independently from the current case. Another detail worth citing is that there is no INTEGER type defining constants in core ASN.1 (see section 2.1, steps 2 and 12).

$$\begin{aligned}
&[\![\text{`}\mathsf{TRef}\,(x)]\!]_\alpha(\mathcal{Q}_\mathcal{R}) = \\
&\quad \textbf{if}\ x \in \mathcal{R}\ \textbf{then}\ \alpha \doteq \mathcal{Q}(x) \\
&\quad \textbf{else match}\ \Gamma(x)\ \textbf{with} \\
&\qquad (\mathrm{T}_0, \mathsf{None}) \rightarrow\ [\![\mathrm{T}_0]\!]_\alpha(\mathcal{Q} \uplus \{x \mapsto \alpha\}) \\
&\qquad \mid (\mathrm{T}_0, \mathsf{Some}\,\mathrm{C}_0) \rightarrow [\![\mathrm{T}_0, \mathrm{C}_0]\!]_\alpha(\mathcal{Q} \uplus \{x \mapsto \alpha\})
\end{aligned}$$

This equation defines the constraint collected from a type reference. Two situations can occur. First, if the reference name x is already in

the domain $\mathcal{R}$ of the mapping $\mathcal{Q}$, it means that we already analysed this type before. Then we just emit an $\alpha \doteqdot \mathcal{Q}(x)$ constraint that expresses that the terms of the reference α are exactly the terms of the referenced type $\mathcal{Q}(x)$. Otherwise, we analyse the referenced type: it can be either a type (first pattern) or a subtype (second pattern).

In the first case, we just analyse the referenced type, without forgetting to add ($\uplus$) a binding $x \mapsto \alpha$ to $\mathcal{Q}$, in order to avoid a loop at run-time in presence of recursive types[6]. A valid situation can be simply illustrated by the declaration `T ::= CHOICE {a REAL, b T}`.

In the second case, we use the constraint collection from subtypes, presented in the next section. A valid situation is: `T ::= SET {a U} U ::= REAL (0..5)`. This case is the only dependence between constraints from types and constraints from subtypes.

Let us consider now an example of constraint collection from a type. Let `T ::= CHOICE {item INTEGER, and SET OF T, not T}`. We want the constraint whose solution in α is the set of terms of `T`, that is to say, $[\![\text{`TRef ("T")}]\!]_\alpha(\{\}) = [\![\textsc{Choice}\,\mathrm{F}_\mathcal{L}]\!]_\alpha(\mathcal{Q})$, where $\mathcal{L} = \{\text{"item"}, \text{"and"}, \text{"not"}\}$ and $\mathcal{Q} = \{\text{"T"} \mapsto \alpha\}$.

Let $\begin{cases} \mathrm{F}\,(\text{"item"}) = \textsc{Integer} \\ \mathrm{F}\,(\text{"and"}) = \textsc{Set of}\,(\text{`TRef ("T")}) \\ \mathrm{F}\,(\text{"not"}) = \text{`TRef ("T")} \end{cases}$

Then we get:

$$
\begin{aligned}
[\![\textsc{Choice}\,\mathrm{F}_\mathcal{L}]\!]_\alpha(\mathcal{Q}) &= [\![\textsc{Choice}\,\mathrm{F}_{\mathcal{L}\setminus\{\text{"item"}\}}]\!]_\gamma(\mathcal{Q}) \\
&\ddot{\wedge}\ [\![\mathrm{F}\,(\text{"item"})]\!]_\beta(\mathcal{Q}) \\
&\ddot{\wedge}\ \alpha \doteqdot (\text{"item"} : \beta) \,\dot{\cup}\, \gamma \\
[\![\textsc{Choice}\,\mathrm{F}_{\mathcal{L}\setminus\{\text{"item"}\}}]\!]_\gamma(\mathcal{Q}) &= [\![\textsc{Choice}\,\mathrm{F}_{\{\text{"not"}\}}]\!]_\delta(\mathcal{Q}) \\
&\ddot{\wedge}\ [\![\mathrm{F}\,(\text{"and"})]\!]_\varepsilon(\mathcal{Q}) \\
&\ddot{\wedge}\ \gamma \doteqdot (\text{"and"} : \varepsilon) \,\dot{\cup}\, \delta \\
[\![\textsc{Choice}\,\mathrm{F}_{\{\text{"not"}\}}]\!]_\delta(\mathcal{Q}) &= [\![\mathrm{F}\,(\text{"not"})]\!]_\zeta(\mathcal{Q}) \\
&\ddot{\wedge}\ \delta \doteqdot \text{"not"} : \zeta \\
[\![\mathrm{F}\,(\text{"item"})]\!]_\beta(\mathcal{Q}) &= [\![\textsc{Integer}]\!]_\beta(\mathcal{Q}) \\
&= \beta \doteqdot \mathbb{N} \\
[\![\mathrm{F}\,(\text{"and"})]\!]_\varepsilon(\mathcal{Q}) &= [\![\textsc{Set of}\,(\text{`TRef ("T")})]\!]_\varepsilon(\mathcal{Q}) \\
&= [\![\text{`TRef ("T")}]\!]_\eta(\mathcal{Q}) \\
&\quad \ddot{\wedge}\ \varepsilon \doteqdot \theta \diamond \mathbb{N}^+ \\
&\quad \ddot{\wedge}\ \theta \doteqdot \text{`Cons}\,(\eta, \theta) \,\dot{\cup}\, \text{`Nil} \\
[\![\mathrm{F}\,(\text{"not"})]\!]_\zeta(\mathcal{Q}) &= [\![\text{`TRef ("T")}]\!]_\zeta(\mathcal{Q}) \\
&= \zeta \doteqdot \alpha \\
[\![\text{`TRef ("T")}]\!]_\eta(\mathcal{Q}) &= \eta \doteqdot \alpha
\end{aligned}
$$

After substitution, we get:

$$
\begin{aligned}
&[\![\text{`TRef ("T")}]\!]_\alpha(\{\}) = \\
&\qquad \zeta \doteqdot \alpha \ \ddot{\wedge}\ \delta \doteqdot \text{"not"} : \zeta \ \ddot{\wedge}\ \eta \doteqdot \alpha \\
&\quad \ddot{\wedge}\ \varepsilon \doteqdot \theta \diamond \mathbb{N}^+ \ \ddot{\wedge}\ \theta \doteqdot \text{`Cons}\,(\eta, \theta) \,\dot{\cup}\, \text{`Nil} \\
&\quad \ddot{\wedge}\ \gamma \doteqdot (\text{"and"} : \varepsilon) \,\dot{\cup}\, \delta \ \ddot{\wedge}\ \beta \doteqdot \mathbb{N} \\
&\quad \ddot{\wedge}\ \alpha \doteqdot (\text{"item"} : \beta) \,\dot{\cup}\, \gamma
\end{aligned}
$$

This constraint implies the system (we do not define formally this

[6]Hence there is no fixpoint-based approach in the collection, it lies in the solving procedure instead.

notion in this paper, because we rely on the solving algorithm of Aiken and Wimmers [12]):

$$\begin{cases} \alpha \doteq (\texttt{"item"} : \beta) \dot\cup (\texttt{"and"} : \varepsilon) \dot\cup (\texttt{"not"} : \alpha) \\ \beta \doteq \mathbb{N} \\ \varepsilon \doteq \theta \diamond \mathbb{N}^+ \\ \theta \doteq \text{`}\mathsf{Cons}\,(\alpha, \theta) \dot\cup \text{`}\mathsf{Nil} \end{cases}$$

The following examples is an illegal type definition. In section 2.2 (validation) we mentionned: `T ::= REAL (ALL EXCEPT T)`. We want the constraint whose solution in α is the set of terms of `T`:

$$[\![\text{`}\mathsf{TRef}\,(\texttt{"T"})]\!]_\alpha(\{\}) =$$
$$[\![\textsc{Real}, \textsc{All except}\,(\textsc{Includes}\,\texttt{"T"})]\!]_\alpha(\{\texttt{"T"} \mapsto \alpha\})$$

Let $\mathcal{Q} = \{\texttt{"T"} \mapsto \alpha\}$. Then we get:

- $[\![\textsc{Real}]\!]_\beta(\mathcal{Q}) = \beta \doteq \mathbb{R}$
- $[\![\textsc{Real}, \textsc{Includes}\,(\texttt{"T"})]\!]_\gamma(\mathcal{Q})$
  $= [\![\textsc{Real}]\!]_\delta(\mathcal{Q}) \ddot\wedge [\![\text{`}\mathsf{TRef}\,(\texttt{"T"})]\!]_\varepsilon(\mathcal{Q}) \ddot\wedge \gamma \doteq \delta \dot\cap \varepsilon$
  $= \delta \doteq \mathbb{R} \ddot\wedge \varepsilon \doteq \alpha \ddot\wedge \gamma \doteq \delta \dot\cap \varepsilon$
- $[\![\text{`}\mathsf{TRef}\,(\texttt{"T"})]\!]_\alpha(\{\})$
  $= \quad [\![\textsc{Real}]\!]_\beta(\mathcal{Q}) \ddot\wedge \alpha \doteq \beta \dot\setminus \gamma$
  $\ddot\wedge \; [\![\textsc{Real}, \textsc{Includes}\,(\texttt{"T"})]\!]_\gamma(\mathcal{Q})$
  $= \quad \beta \doteq \mathbb{R} \ddot\wedge \delta \doteq \mathbb{R} \ddot\wedge \alpha \doteq \beta \dot\setminus \gamma$
  $\ddot\wedge \; \varepsilon \doteq \alpha \ddot\wedge \gamma \doteq \delta \dot\cap \varepsilon$

This constraint implies (we do not define formally this notion in this paper): $\alpha \doteq \dot\neg\alpha$, which has no solutions. Thus the declaration of subtype `T` must be rejected.

6 Constraints from Subtypes

In this section we define the collection of constraints from subtypes, *i.e.,* on the declarations of pattern <type reference> ::= <non-reference type without inner constraints> (<subtyping constraint>). It is a function of type $\mathcal{T} \times \mathcal{C} \to V \to (\mathcal{R} \to V) \to \mathcal{K}$, where $\mathcal{C}$ is the set of subtyping constraints. We use a similar notation for this mapping as in section 5, *e.g.,* $[\![\mathrm{T}, \mathrm{C}]\!]_\alpha(\mathcal{Q}_\mathcal{R})$.

6.1 Regular expression constraint

The strings can be constrained to belong to a regular language defined by means of a regular expression (similarly to the Perl scripting language, or the XML, or the grep Unix command) introduced by the Pattern keyword. The following specifies date and time in format 'dd/mm/yyyy-hh:mm': `DateAndTime ::= VisibleString(PATTERN "\d#2/\d#2/\d#4-\d#2:\d#2")`. Formally, the semantics is expressed as follows:

$$[\![\mathsf{String}, \textsc{Pattern}\ (s)]\!]_\alpha(\mathcal{Q}) = \alpha \doteq \text{`}\mathsf{Regexp}\,(s)$$

This equation defines the constraint collected from a regular expression constraint (introduced by the Pattern keyword). This case is

straightforward since we have a built-in notion of regular expression (‘Regexp).

6.2 Union constraint

Given two constraints, it is possible to create a new constraint that is the union of both, using the keyword `UNION` or the symbol ‘`|`’. The semantics is then that the new subtype contains the values of the first subtype and of the second subtype. For instance, let us define `Day ::= ENUMERATED {monday, tuesday, wednesday, thursday, friday, saturday, sunday}`. Then: `WeekEnd ::= Day (saturday UNION sunday)`.

$[\![\mathrm{T}, \mathrm{C}_0 \textsc{ Union } \mathrm{C}_1]\!]_\alpha(\mathcal{Q}) =$
let β *and* γ *be fresh variables*
in $[\![\mathrm{T}, \mathrm{C}_0]\!]_\beta(\mathcal{Q}) \ddot{\wedge} [\![\mathrm{T}, \mathrm{C}_1]\!]_\gamma(\mathcal{Q}) \ddot{\wedge} \alpha \ddot{=} \beta \dot{\cup} \gamma$

This equation defines the collection of constraints from a union of subtyping constraints, C_0 and C_1, which apply to a type T. First, we collect the constraints from the subtypes $(\mathrm{T}, \mathrm{C}_0)$ and $(\mathrm{T}, \mathrm{C}_1)$. The associated sets are respectively β and γ, and α is $\beta \dot{\cup} \gamma$, as expected.

6.3 Intersection constraint

Given two subtyping constraints, we can create a new one which is the intersection of both, using the `INTERSECTION` keyword or the symbol ‘`^`’. The semantics is that the new subtype contains only the values that belong to the two subtypes. For instance, we can define a type for the French telephone numbers: `PhoneNumber ::= NumericString ((FROM("0".."9")) INTERSECTION (SIZE(10)))`, using an alphabet constraint (section 6.6) and a size constraint (section 6.7). The formal semantics is as follows:

$[\![\mathrm{T}, \mathrm{C}_0 \textsc{ Intersection } \mathrm{C}_1]\!]_\alpha(\mathcal{Q}) =$
let β *and* γ *be fresh variables*
in $[\![\mathrm{T}, \mathrm{C}_0]\!]_\beta(\mathcal{Q}) \ddot{\wedge} [\![\mathrm{T}, \mathrm{C}_1]\!]_\gamma(\mathcal{Q}) \ddot{\wedge} \alpha \ddot{=} \beta \dot{\cap} \gamma$

It is the dual case of section 6.2.

6.4 Inclusion constraint

It is possible to restrict a subtype to only have the values of a given subtype, using the `INCLUDES` keyword. For instance, following the example given in section 6.2, we can define: `LongWeekEnd ::= Day (INCLUDES WeekEnd | monday)`, or, as a short-hand `LongWeekEnd ::= Day (WeekEnd | monday)`. Formally, the semantics is

$[\![\mathrm{T}, \textsc{Includes } \mathrm{T}']\!]_\alpha(\mathcal{Q}) =$
let β *and* γ *be fresh variables*
in $[\![\mathrm{T}]\!]_\beta(\mathcal{Q}) \ddot{\wedge} [\![\mathrm{T}']\!]_\gamma(\mathcal{Q}) \ddot{\wedge} \alpha \ddot{=} \beta \dot{\cap} \gamma$

The values of the type T are restricted to be in the set of values of the type T’. Note that, since we work in core ASN.1, the type T’

is a reference (section 2.1, step 5). This case is very similar to the intersection constraint presented in section 6.3, and naturally has a very similar semantics.

6.5 Exclusion constraint

The protocol designer can restrict the values of a type to *not* belong to another subtype, by means of a constraint preceded by `ALL EXCEPT`, or two constraints separated by `EXCEPT`. Consider for instance `Lipogram ::= IA5String (FROM (ALL EXCEPT ("e" | "E")))`, defining the set of strings which do not contain the characters `"e"` and `"E"`. Formally, we have

$$[\![\mathrm{T}, \mathrm{C}_0 \text{ Except } \mathrm{C}_1]\!]_\alpha(\mathcal{Q}) = \\ [\![\mathrm{T}, \mathrm{C}_0 \text{ Intersection } (\text{All except } \mathrm{C}_1)]\!]_\alpha(\mathcal{Q})$$

This first equation defines the semantics of `EXCEPT` using the `ALL EXCEPT` constraint. The underlying rationale is the following equality on sets: $\beta \backslash \gamma = \beta \cap (\alpha \backslash \gamma)$, for all $\beta \subseteq \alpha$ and $\gamma \subseteq \alpha$. The remaining situation is

$$[\![\mathrm{T}, \text{All except } \mathrm{C}_0]\!]_\alpha(\mathcal{Q}) = \\ \textbf{let } \beta \textit{ and } \gamma \textit{ be fresh variables} \\ \textbf{in } [\![\mathrm{T}]\!]_\beta(\mathcal{Q}) \ \ddot{\wedge} \ [\![\mathrm{T}, \mathrm{C}_0]\!]_\gamma(\mathcal{Q}) \ \ddot{\wedge} \ \alpha \doteq \beta \dot{\setminus} \gamma$$

6.6 Alphabet constraint

The strings can be restricted to be built upon a given alphabet, using the keyword `FROM`. For instance, we can define a subtype whose values are strings made of capital and small letters: `CapitalAndSmall ::= IA5String (FROM ("A".."Z" | "a".."z"))`. Contrast with the following: `CapitalOrSmall ::= IA5String (FROM("A".."Z") | FROM ("a".."z"))`. These examples combine an alphabet constraint and an interval constraint (section 6.8).

$$[\![\mathsf{String}, \text{From } \mathrm{C}_0]\!]_\alpha(\mathcal{Q}) = \\ \textbf{let } \beta \textit{ be a fresh variable } \textbf{in} \\ \quad \textbf{let } \mathrm{C}' = (\text{Size } (\text{`PosInt } 1)) \text{ Intersection } \mathrm{C}_0 \textbf{ in} \\ \quad \textbf{let } \kappa = [\![\mathsf{String}, \mathrm{C}']\!]_\beta(\mathcal{Q}) \\ \textbf{in match } \textit{solve\_regexp } (\kappa)\,(\beta) \textbf{ with} \\ \quad \mathsf{Some}\,(s) \rightarrow \alpha \doteq \text{`Regexp}\,(s)$$

This equation defines the constraint collection from the alphabet subtyping constraint. The constraint C_0 models a set of strings allowed by T. We construct a subtyping constraint C' that defines the *characters* of C_0 (characters are strings of size one) in order to build the corresponding semantic constraint κ. We must resolve it at this stage because we want to use a regexp constraint to model the alphabet subtyping constraint. In other words, we need to compute a string following the ASN.1 regular expression syntax and representing κ. This is done by the function $\textit{solve\_regexp} : \mathcal{K} \rightarrow V \rightarrow \textit{string option}$ which takes as arguments a constraint denoting an alphabet, and a variable occurring in this constraint.

First, *solve_regexp* $(\kappa)\,(\beta)$ constructs the set of constraints which are the conjuncts of κ. By analysing all the cases of our algorithm (sections 5 and 6), it turns out that their patterns must be of five kinds only: $\alpha \doteqdot \beta \dot{\cup} \gamma$, $\alpha \doteqdot \beta \dot{\cap} \gamma$, $\alpha \doteqdot \beta \dot{\setminus} \gamma$, $\alpha \doteqdot$ ‘Regexp (r) or $\alpha \doteqdot \dot{\bigcup}_{1 \leqslant i \leqslant n}$ ‘Regexp (r_i), otherwise we reject the constraint as inconsistent. The last kind is transformed into the equivalent: $\alpha \doteqdot$ ‘Regexp "(r_1) | (r_2) | ... | (r_n)". These constraints form a non-recursive system, which can be solved in β by simple substitution. The last step is to compute strings denoting intersections, unions and complements of regular expressions (the result is always a regular expression because regular expressions are closed under such operations). Consider now the following example. Let

$$\kappa = (\alpha \doteqdot \beta \dot{\cup} \dot{\neg}\gamma) \;\ddot{\wedge}\; (\beta \doteqdot \text{‘Regexp "[0-9A-F]*"}) \;\ddot{\wedge}\; (\gamma \doteqdot \text{‘Regexp "[01]*"}),$$

then *solve_regexp* $(\kappa)\,(\alpha)$ = Some "([0-9A-F]*) | (^[01]*)". If the return value is None, then it is an inconsistency.

6.7 Size constraint

The values of string types can be constrained to have given sizes, by introducing a subtyping constraint by the keyword `SIZE`. For instance:

```
Exactly31BitsString ::= BIT STRING (SIZE (31))
StringOf31BitsAtTheMost::= BIT STRING(SIZE(0..31))
NonEmptyString ::= OCTET STRING (SIZE (1..MAX))
```

The size constraint can also apply to SET OF and SEQUENCE OF types. In that case, the semantics is very different: the values of the types are sets whose *cardinals* are specified by the size constraint. Moreover, the constraint must appear between the keywords `SET` or `SEQUENCE`, and `OF`, as in `ListOf5Strings ::= SEQUENCE (SIZE (5)) OF PrintableString`. Contrast this with `ListOfStringsOf5Char::= SEQUENCE OF PrintableString (SIZE (5))`, where the strings themselves are constrained, not the cardinal of the SEQUENCE OF. See Figure 1.

This equation defines the constraint collected from a size subtyping constraint C_0. The function *apply* takes as its first argument a string *kind*, and a positive (closed) interval as its second argument. The kind is a string that determines the alphabet of the string T: "[01]" for bit strings, "[0-9A-F]" for octet strings, and "." for general strings. The interval is used to build an ASN.1 regular expression determining the repetition of the characters in the kind. Both are finally combined by *apply*. For example: *apply* ("[01]") (‘PosInt (3), ‘PlusInfInt) = "([01]#(3,))", for bit strings whose size is at least three.

We can construct a subtyping constraint C which is a restriction of C_0 to the positive integers (indeed, a size must be a positive integer). Then we can compute the constraint corresponding to the sizes: ‘**let** $\kappa = [\![\text{INTEGER}, C]\!]_\delta(\mathcal{Q})$’. Our aim is to use the regexp constraint expressions to model the SIZE subtyping constraint, hence we need to extract from κ a suitable set of constraints: a *disjunctive normal form*

$$
\begin{array}{l}
[\![\mathrm{T}, \textsc{Size}\ \mathrm{C}_0]\!]_\alpha(\mathcal{Q}) = \\
\quad \textsf{let}\ \ \beta, \gamma \ \textit{and}\ \delta \ \textit{be fresh variables} \\
\quad \textsf{and}\ \textit{apply}\ (\textit{kind} : \textit{string})\ (\text{`}\mathsf{Interval}\ (\text{`}\mathsf{PosInt}(n), \textit{ub})) = \\
\qquad \textsf{let}\ \textit{up} = \textsf{match}\ \textit{ub}\ \textsf{with}\ \text{`}\mathsf{PlusInfInt} \to \text{""} \\
\qquad\qquad\qquad \mid \text{`}\mathsf{PosInt}\ (m) \to \textit{string\_of\_int}\ (m) \\
\qquad \textsf{in}\ \ \text{"("} \mathbin{\hat{}}\ \textit{kind} \mathbin{\hat{}}\ \text{"\#("} \mathbin{\hat{}}\ \textit{string\_of\_int}\ (n) \\
\qquad\quad \mathbin{\hat{}}\ \text{","} \mathbin{\hat{}}\ \textit{up} \mathbin{\hat{}}\ \text{"))"} \\
\quad \textsf{and}\ \mathrm{C} = (\text{`}\mathsf{PosInt}\ (0) \leqslant .. < \textsc{Max}) \\
\qquad\qquad \textsc{Intersection}\ \mathrm{C}_0\ \textsf{in} \\
\quad \textsf{let}\ \kappa = [\![\textsc{Integer}, \mathrm{C}]\!]_\delta(\mathcal{Q}) \\
\quad \textsf{in}\ \textsf{match}\ \textit{solve\_integers}\ (\kappa)\ (\delta) \\
\qquad \textsf{with}\ \textit{dnf}\ \textsf{when}\ \textit{dnf} \neq \varnothing\ \&\&\ \textit{dnf} \neq \{\dot{0}\} \to \\
\qquad\ (\textsf{match}\ \mathrm{T}\ \textsf{with}\ \textsc{Bit string} \to \\
\qquad\qquad \alpha \doteqdot \dot{\bigcup}_{\varsigma \in \textit{dnf}}\ \text{`}\mathsf{Regexp}\ (\textit{apply}\ \text{"[01]"}\ (\varsigma)) \\
\qquad\quad \mid \textsc{Octet string} \to \\
\qquad\qquad \alpha \doteqdot \dot{\bigcup}_{\varsigma \in \textit{dnf}}\ \text{`}\mathsf{Regexp}\ (\textit{apply}\ \text{"[0-9A-F]"}\ (\varsigma)) \\
\qquad\quad \mid \mathsf{String} \to \alpha \doteqdot \dot{\bigcup}_{\varsigma \in \textit{dnf}}\ \text{`}\mathsf{Regexp}\ (\textit{apply}\ \text{"."}\ (\varsigma)) \\
\qquad\quad \mid \textsc{Set of}\ \mathrm{T}_0 \mid \textsc{Sequence of}\ \mathrm{T}_0 \to [\![\mathrm{T}_0]\!]_\beta(\mathcal{Q})\ \ddot{\wedge} \\
\qquad\qquad \gamma \doteqdot \text{`}\mathsf{Cons}\ (\beta, \gamma)\ \dot{\cup}\ \text{`}\mathsf{Nil}\ \ddot{\wedge}\ \alpha \doteqdot \dot{\bigcup}_{\varsigma \in \textit{dnf}} \gamma \diamond \varsigma)
\end{array}
$$

Figure 1: Size constraint

which is the smallest union of closed intervals. This is the purpose of $\textit{solve\_integers} : \mathcal{K} \to V \to [> \textit{closed\_int\_interval} \mid \dot{0}]\ \textit{set}$ (as a shorthand, we assume that we have an abstract polymorphic type $'\mathsf{a}\ \textit{set}$). The returned value of $\textit{solve\_integers}\ (\kappa)\ (\delta)$ is called $\textit{dnf}$, where δ is the unknown in κ we are interested in. If it is $\varnothing$, it means an inconsistency error (note that the property $\dot{0} \in \textit{dnf} \Rightarrow \textit{dnf} = \{\dot{0}\}$ always holds). The analysis of all the cases of our algorithm shows that each conjunct in κ must have the pattern $\alpha \doteqdot \beta \,\dot{\cup}\, \gamma$, $\alpha \doteqdot \beta \,\dot{\cap}\, \gamma$, $\alpha \doteqdot \beta \,\dot{\setminus}\, \gamma$ or $\alpha \doteqdot \text{`}\mathsf{Interval}\ (\textit{lb}, \textit{ub})$. They form a non-recursive system of equations on closed intervals, whose left-hand side variables are unique, hence resolution by substitution is straightforward.

The type T can be a string. In this case, the constraint is built as the union ($\dot{\cup}$) of the constraints (`Regexp ...) associated (according to the actual kind of T) to each interval (ς) of sizes ($\textit{dnf}$).

Otherwise, T is actually either a Set of or a Sequence of type, and the semantics is very different. There is no easy way to encode sets whose sizes (cardinals) range over an interval with our constraints, *e.g.,* a Set value whose size is 3.000, encoded as the embedding of 3.000 `Cons constructors. So we decide to issue a powerset constraint made of the expression collected from T and the actual cardinals of its elements. More precisely, we return a constraint similar to the constraint $[\![\mathrm{T}]\!]_\alpha(\mathcal{Q})$ (see section 5), but whose powerset constraint is $\dot{\bigcup}_{\varsigma \in \textit{dnf}} \gamma \diamond \varsigma$ instead of $\gamma \diamond \mathbb{N}^+$. As an example, let us consider now the declaration:

`A ::= SET (SIZE (3..8|7..10|12)) OF REAL`. Then the constraint collected from the type `A` is:

$$[\![\text{`TRef "A"}]\!]_\alpha(\{\}) = [\![\text{Set of Real}]\!]_\beta(\text{"A"} \mapsto \alpha)$$
$$\ddot{\wedge}\ \alpha \doteqdot \beta\Diamond\text{`Interval}\,(\text{`PosInt}\,(3), \text{`PosInt}\,(10))$$
$$\dot{\cup}\ \beta\Diamond\text{`Interval}\,(\text{`PosInt}\,(12), \text{`PosInt}\,(12)).$$

6.8 Interval constraint

The Integer, Real and (almost all) string types have totally ordered values, hence allowing interval definitions for their values. Consider for instance

```
PositiveOrZeroInteger ::= INTEGER (0..MAX)
PositiveInteger ::= INTEGER (0<..MAX)
NegativeOrZeroInteger ::= INTEGER (MIN..0)
NegativeInteger ::= INTEGER (MIN..<0)
PositiveReal ::= REAL (0<..PLUS-INFINITY)
NegativeReal ::= REAL (MINUS-INFINITY..<0)
RealInterval ::= REAL (4e-5..1e-4)
```

```
PositiveOrZeroInteger ::= INTEGER (0..MAX)
PositiveInteger ::= INTEGER (0<..MAX)
NegativeOrZeroInteger ::= INTEGER (MIN..0)
NegativeInteger ::= INTEGER (MIN..<0)
PositiveReal ::= REAL (0<..PLUS-INFINITY)
NegativeReal ::= REAL (MINUS-INFINITY..<0)
RealInterval ::= REAL (4e-5..1e-4)
```

The formal semantics of this kind of constraint is:

$$[\![\mathrm{T}, v_0\, b_0 \,..\, b_1\, v_1]\!]_\alpha(\mathcal{Q}) =$$
let *norm_int_interval* :
interval → *closed_int_interval* = ...
and *norm_real_interval* :
interval → *real_interval* = ...
and *regexp_of_str_interval* :
interval → [>‘Regexp **of** *string*] = ...
in match T **with**
Integer → $\alpha \doteqdot$ *norm_int_interval*$(v_0\, b_0 \,..\, b_1\, v_1)$
| Real → $\alpha \doteqdot$ *norm_real_interval*$(v_0\, b_0 \,..\, b_1\, v_1)$
| String → $\alpha \doteqdot$ *regexp_of_str_interval*$(v_0\, b_0 \,..\, b_1\, v_1)$

This equation defines the constraint collection from value range subtyping constraints, in other words, interval constraints. These constraints can be applied to the Integer, Real and String types, because the values of these types can be totally ordered. So, according to the actual type, the specified value range constraint will be transformed respectively into an OCaml value of type *closed_int_interval* (*i.e.,* a closed integer interval), *real_interval*, or [> ‘Regexp **of** *string*] (that is to say, it is a regular expression). Due to the lack of room, we cannot give the pieces of codes of the three functions implementing these transformations, but there is nothing really difficult here.

6.9 Value constraint

It is possible to restrict the set of values of a type to be a singleton, by simply specifying this unique value between parenthesis. Consider again the example of section 6.2: `Wednesday ::= Day (wednesday)`, and the type `LongWeekEnd` in section 6.4. As we showed in section 2.1 at step 3, all the declared values in core ASN.1 appear in value constraints (of their expected type). Therefore, this section provides the solution to the type compatibility problem (section 2.2, item 3). Typically, we want to compute the constraint $[\![\mathrm{T}, \text{`}\mathsf{VRef}(y_0)]\!]_\alpha(\mathcal{Q})$, with the pseudo-specification excerpt

$$\mathtt{y}_0\ \mathtt{X}_1\ \mathtt{::=}\ \mathtt{y}_1 \quad \mathtt{y}_1\ \mathtt{X}_2\ \mathtt{::=}\ \mathtt{y}_2\ \ldots \quad \mathtt{y}_{n-1}\ \mathtt{X}_n\ \mathtt{::=}\ v\,\mathtt{,}$$

where v is not a value reference. We have to unfold the reference $\mathtt{y}_0$ and get: $[\![\mathrm{T}, \text{`}\mathsf{VRef}(y_0)]\!]_\alpha(\mathcal{Q}) = [\![\mathrm{T}, v]\!]_\alpha(\mathcal{Q})$. Moreover, the semantic model of ASN.1 implies that the following condition must hold:

$$\begin{aligned}[\![\mathrm{T}]\!]_\beta(\{\}) &= [\![\text{`}\mathsf{TRef}\ (\mathtt{"X}_1\mathtt{"})]\!]_\beta(\{\}) = [\![\text{`}\mathsf{TRef}\ (\mathtt{"X}_2\mathtt{"})]\!]_\beta(\{\}) = \ldots \\ &= [\![\text{`}\mathsf{TRef}\ (\mathtt{"X}_n\mathtt{"})]\!]_\beta(\{\}),\end{aligned}$$

i.e., all the types must have the same value sets. For instance, the core ASN.1 specification:

$$\begin{cases} \mathtt{a\ A\ ::=\ b} & \mathtt{A\ ::=\ SET\ \{x\ REAL\ OPTIONAL\}} \\ \mathtt{b\ B\ ::=\ 0.0} & \mathtt{B\ ::=\ SET\ \{x\ REAL\}} \end{cases}$$

must be rejected because the value `{}` does not belong to the type `B`, despite `0.0` belonging to both `A` and `B`. The following equation formally defines this semantics:

$$\begin{array}{l}
[\![\mathrm{T}, \text{`}\mathsf{VRef}\ (y_0)]\!]_\alpha(\mathcal{Q}) = \\
\quad \textbf{let}\ \beta\ \textit{be a fresh variable}\ \textbf{in} \\
\quad\quad \textbf{let rec}\ \textit{unfold}\ (f : \mathcal{V} \to \mathcal{K}) = \ \textbf{function} \\
\quad\quad\quad ((x_1, \text{`}\mathsf{VRef}\ (y_1)) : \mathcal{R} \times v_ref) \to \\
\quad\quad\quad\quad \textbf{let}\ f'\,(v) = f(v)\ \ddot{\curlywedge}\ [\![\text{`}\mathsf{TRef}\ (x_1)]\!]_\beta(\{\}) \\
\quad\quad\quad\quad \textbf{in}\ \ \textit{unfold}\ \ f'\,(\Delta(y_1)) \\
\quad\quad\quad | \ (\underline{\quad}, v) \to f\,(v) \\
\quad\quad \textbf{in}\ \overline{\textit{unfold}}\ (\textbf{fun}\ v \to [\![\mathrm{T}, v]\!]_\alpha(\{\})\ \ddot{\curlywedge}\ [\![\mathrm{T}]\!]_\beta(\{\}))\ (\Delta(y_0))
\end{array}$$

If the value v is not a reference, the following equations apply, in accordance with the type T:

$$\begin{array}{l}
[\![(\textsc{Set of}\ \underline{\quad} \mid \textsc{Sequence of}\ \underline{\quad}), v]\!]_\alpha(\mathcal{Q}) = \\
\quad \textbf{let}\ \overline{\textit{length}} = \textbf{match}\ v\ \textbf{with} \\
\qquad\qquad\qquad \text{`}\mathsf{List}\,(\mathrm{L}) \to \mathrm{List}.\textit{length}\,(\mathrm{L}) \\
\qquad\qquad\quad | \ \text{`}\mathsf{Nil} \to 0 \\
\quad \textbf{in}\ \ \alpha \doteq \mu(v) \diamond \mu(\text{`}\mathsf{PosInt}(\textit{length}))
\end{array}$$

This equation applies for Set of and Sequence of types. The value v must be either ‘List or ‘Nil. We compute the length *length* of v. Hence $\mu\,(\text{`}\mathsf{PosInt}\,(\textit{length}))$ is the expression corresponding to a closed integer interval containing only the size of v. Finally, we form the powerset expression $\mu(v) \diamond \mu(\ldots)$.

- $[\![\text{REAL}, v]\!]_\alpha(\mathcal{Q}) = \ \alpha \doteqdot \mu\,(\mathit{normalise\_real}\,(v))$
- $[\![\text{INTEGER}, v]\!]_\alpha(\mathcal{Q}) = \ \alpha \doteqdot \mu\,(v)$
- $[\![(\text{BIT STRING} \mid \text{OCTET STRING} \mid \mathsf{String}), v]\!]_\alpha(\mathcal{Q}) = \alpha \doteqdot \mu\,(v)$
- $[\![\mathrm{T}, v]\!]_\alpha(\mathcal{Q}) = \ \alpha \doteqdot \mu\,(v)$ otherwise.

These equations handle the remaining cases. If T is REAL, we need to normalise the value v by means of the *normalise_real* function (whose code is not shown here). It consists of rewriting the values of the type associated to REAL using the decimal (dotted) notation: `{mantissa 1, base 10, exponent -3}` $\longrightarrow$ `1E-3`, and each integer literal of the REAL type is rewritten with the decimal notation: `5` $\longrightarrow$ `5.0`. Then μ is applied: $\mu\,(\mathit{normalise\_real}\,(v))$. The remaining equations are obvious.

6.10 Inner type constraints

When some SET OF or SEQUENCE OF type is imported from another ASN.1 module, we cannot syntactically insert a constraint on its elements between the keywords `SET` or `SEQUENCE` and `OF` (see section 6.7). That is why there is another way of achieving the same goal: using the `WITH COMPONENT` constraint. For instance, quoting Dubuisson [7, §13.8.11], given `TextBlock ::= SEQUENCE OF VisibleString`, we can further refine this type for lines of no more than 32 characters: `AddressBlock ::= TextBlock (WITH COMPONENT (SIZE (1..32)))`, or for digits and spaces: `DigitBlock ::= TextBlock (WITH COMPONENT (NumericString))`. These two subtypes have the same values as: `AddressBlock ::= SEQUENCE OF VisibleString(SIZE(1..32))` and `DigitBlock ::=SEQUENCE OF VisibleString(NumericString)`. Formally:

$$
\begin{array}{l}
[\![(\text{SET OF } \mathrm{T}_0 \mid \text{SEQUENCE OF } \mathrm{T}_0),\\
\quad \text{WITH COMPONENT } \mathrm{C}_0]\!]_\alpha(\mathcal{Q}) =\\
\quad \textbf{let } \beta \textit{ and } \gamma \textit{ be fresh variables}\\
\quad \textbf{in } \ [\![\mathrm{T}_0, \mathrm{C}_0]\!]_\beta(\mathcal{Q}) \ \ddot{\wedge} \ \gamma \doteqdot \text{`}\mathsf{Cons}\,(\beta, \gamma) \,\dot{\cup}\, \text{`}\mathsf{Nil}\\
\qquad \ddot{\wedge} \ \alpha \doteqdot \gamma \diamond \mathbb{N}^+
\end{array}
$$

This equation defines the constraint collection from the WITH COMPONENT subtyping constraint. It applies to SET OF and SEQUENCE OF types, corresponding to powersets of values, by constraining their elements (but not their cardinals). For instance `A ::= SET (WITH COMPONENT (0..7)) OF INTEGER` is the type of the set of integers ranging from 0 to 7. First, we extract the expression corresponding to the type of the elements *as constrained by the clause* WITH COMPONENT, and bind it to the fresh variable β: $[\![\mathrm{T}_0, \mathrm{C}_0]\!]_\beta(\mathcal{Q})$. Then we form the constraint $\gamma \doteqdot \text{`}\mathsf{Cons}\,(\beta, \gamma) \,\dot{\cup}\, \text{`}\mathsf{Nil}$ which denotes the powerset over sets of β elements. Finally, we get the powerset constraint $\alpha \doteqdot \gamma \diamond \mathbb{N}^+$ which assigns the set $\mathbb{N}^+$ as the cardinals for the elements of γ.

Now let us introduce the *partial constraints* of SET OF, SEQUENCE and CHOICE types. Given

```
Quadruple ::= SEQUENCE {
  alpha ENUMERATED {in, out} OPTIONAL,
  beta  IA5String OPTIONAL,
  gamma SEQUENCE OF INTEGER,
  delta BOOLEAN DEFAULT TRUE}
```

we can derive a subtype whose component `alpha` is always present and equals `in`, and the component `gamma` always has five elements:

```
Quadruple1 ::= Quadruple (
    WITH COMPONENTS {..., alpha (in) PRESENT,
                          gamma (SIZE (5))})
```

This subtype has the same values as

```
Quadruple1 ::= SEQUENCE {
  alpha ENUMERATED {in, out} (in),
  beta  IA5String OPTIONAL,
  gamma SEQUENCE SIZE (5) OF INTEGER,
  delta BOOLEAN DEFAULT TRUE}
```

The symbol '...' means that we constrain only the listed components: this is a *partial constraint*. If the symbol '...' is missing, there is an implicit constraint `PRESENT` on the listed components, and `ABSENT` on the others:

```
Quadruple1 ::= Quadruple (WITH COMPONENTS
    {alpha (in), gamma (SIZE (5)), delta})
```

This subtype has the same values as

```
Quadruple1 ::= SEQUENCE {
  alpha ENUMERATED {in, out} (in),
  gamma SEQUENCE SIZE (5) OF INTEGER,
  delta BOOLEAN DEFAULT TRUE
}
```

This is called a *complete* or *fully specified* constraint [1, §47.8.6]. The following equation defines the constraint collection from this kind of constraint. They can be expressed in terms of the so-called partial constraints. Let us first examine the case of the CHOICE type:

$$
\begin{array}{l}
[\![\mathrm{T}, \textsc{With components}\ (\mathsf{Full}, \mathrm{K}_{\mathcal{L}'})]\!]_\alpha(\mathcal{Q}) = \\
\quad \textbf{match}\ \mathrm{T}\ \textbf{with} \\
\qquad \textsc{Choice}\ \mathrm{F}_{\mathcal{L}}\ \textbf{when}\ \mathcal{L}' \subseteq \mathcal{L} \rightarrow \\
\qquad\quad \textbf{let}\ \mathrm{A} = \{l \mapsto (\mathsf{None}, \mathsf{Some}\,\textsc{Absent})\}_{l \in \mathcal{L} \setminus \mathcal{L}'}\ \textbf{in} \\
\qquad\qquad \textbf{let}\ \overline{\mathrm{K}}_{\mathcal{L}'} = \mathrm{K}_{\mathcal{L}'} \uplus \mathrm{A} \\
\qquad\quad \textbf{in}\ [\![\mathrm{T}, \textsc{With components}\ (\mathsf{Partial}, \overline{\mathrm{K}}_{\mathcal{L}'})]\!]_\alpha(\mathcal{Q}) \\
\quad \mid \ldots
\end{array}
$$

The component names (*labels*) which carry constraints, $\mathcal{L}'$, must occur in the type definition: $\mathcal{L}' \subseteq \mathcal{L}$ (this is a consistence checking). The fact that the constraint is fully specified implies that the components not explicitly constrained must be absent (A) in the value

notation, and, since moreover the type is a CHOICE, no further constraint is added to the explicitly constrained components. For instance: `A ::= CHOICE {a INTEGER, b T} (WITH COMPONENTS {a (3..10)})` is equivalent to `A ::= CHOICE {a INTEGER, b T} (WITH COMPONENTS {a (3..10), b ABSENT, ...})`.

This semantics may lead to an assumption we made being broken. Indeed, we assumed that the types in core ASN.1 (section 2) are well-founded (section 3), *i.e.,* the types have at least a finite value. This property may not hold here anymore because of this 'component-cancellation' semantics. Consider the following type declaration:

```
A ::= CHOICE {a A, b INTEGER} (WITH COMPONENTS {a}).
```

It is equivalent to `A ::= CHOICE {a A}`, which is obvioulsy not a well-founded type. That is why we must check again this property on the CHOICE with only its explicitly constrained components: $\Gamma \Vdash \text{CHOICE } \mathrm{F}_{\mathcal{L}'}$.

The inner subtyping constraint is represented by $\mathrm{K}_{\mathcal{L}'}$, which is a mapping from labels (belonging to the set $\mathcal{L}'$) to pairs of (possibly optional) subtyping constraint and (possibly optional) presence constraint. For example `WITH COMPONENTS {a (0..9) PRESENT, b ABSENT, c (7)}` corresponds to:

$$\begin{cases} \mathrm{K}(\texttt{"a"}) = (\mathsf{Some}\,(\text{`}\mathsf{PosInt}(0) \leqslant .. \leqslant \text{`}\mathsf{PosInt}(9)), \\ \qquad\qquad \mathsf{Some}\,\textsc{Present}) \\ \mathrm{K}(\texttt{"b"}) = (\mathsf{None}, \mathsf{Some}\,\textsc{Absent}) \\ \mathrm{K}(\texttt{"c"}) = (\mathsf{Some}\,(\text{`}\mathsf{PosInt}(7)), \mathsf{None}) \end{cases}$$

When there is no explicit presence constraint, like for the component `c` in our last example, there is implicitly a PRESENT constraint [1, §48.8.9.2]. This rule is formally specified in the lines '**let** $p = \ldots$ **in let** $\overline{\mathrm{K}}_{\mathcal{L}'} = \ldots$' (let us recall that the notation '$v \lhd p$' is a shorthand for '**match** v **with** $p \to$ **true**').

We can finally create a subtyping constraint that is *partial*, instead of being complete:

$$[\![(\textsc{Choice } \mathrm{F}_{\mathcal{L}'}), \textsc{With components}\,(\mathsf{Partial}, \ldots)]\!]_\alpha(\mathcal{Q}).$$

This way we are able to factorize as much as possible the computations and checks: we reduce the complete constraints to partial ones.

Let us consider the second pattern. It filters the SET and SEQUENCE types:

$$\begin{array}{l} \mid \ldots \\ \mid (\textsc{Set}\,\Phi_{\mathcal{L}} \mid \textsc{Sequence}\,\Phi_{\mathcal{L}}) \textbf{ when } \mathcal{L}' \subseteq \mathcal{L} \to \\ \quad \textbf{let } \mathrm{A} = \{l \mapsto (\mathsf{None}, \mathsf{Some}\,\textsc{Absent})\}_{l \in \mathcal{L} \setminus \mathcal{L}'} \textbf{ in} \\ \quad\quad \textbf{let } \mathrm{P} = \{l' \mapsto (\sigma, \mathsf{Some}\,\textsc{Present}) \mid \\ \quad\quad\quad\quad \mathrm{K}(l') \lhd (\sigma, \mathsf{None})\}_{l' \in \mathcal{L}'} \textbf{ in} \\ \quad\quad \textbf{let } \overline{\mathrm{K}}_{\mathcal{L}'} = \mathrm{K}_{\mathcal{L}'} \uplus \mathrm{A} \uplus \mathrm{P} \\ \quad \textbf{in } [\![\mathrm{T}, \textsc{With components}\,(\mathsf{Partial}, \overline{\mathrm{K}}_{\mathcal{L}'})]\!]_\alpha(\mathcal{Q}) \end{array}$$

First we perform the same consistence checking as for CHOICE types, $\mathcal{L}' \subseteq \mathcal{L}$. As for CHOICE types, since the constraint is fully specified, it implies that the components not explicitly constrained

must be absent in the value notation (A). But, contrary to CHOICE types, the components which are not explicitly constrained are further constrained to be present (P) [1, §47.8.9.3]. For instance: `A ::= SET {a REAL OPTIONAL} (WITH COMPONENTS {a (0.0)})` is equivalent to `A ::= SET {a REAL OPTIONAL} (WITH COMPONENTS {a (0.0) PRESENT, ...})`.

$$[\![\mathrm{T}, \textsc{With components}\ (\mathsf{Partial}, \mathrm{K}_{\mathcal{L}'})]\!]_\alpha(\mathcal{Q}) =$$
match T **with**
CHOICE $\mathrm{F}_{\mathcal{L}}$ **when** $\mathcal{L}' \subseteq \mathcal{L} \rightarrow \ldots$

This equation defines the constraint collection from the partially specified subtyping constraints on CHOICE types. First, we check the consistence $\mathcal{L}' \subseteq \mathcal{L}$. Then we use a non-standard construct **cases** $b_0 \rightarrow e_0 \mid \ldots \mid b_n \rightarrow e_n$ **end**, which means:'**if** b_0 **then** e_0 **else** ... **if** b_n **then** e_n **else fail**':

```
...
cases
    L' = ∅ → [[T]]_α(Q)
  | ∃!l' ∈ L'.(K(l') ◁ (σ, Some PRESENT)
                ∧ Γ ⊩ CHOICE F_{l'}) →
    let  K̄ = {l' ↦ (σ, None)}
    and  T' = CHOICE F_{l'}
    in [[T', WITH COMPONENTS (Partial, K̄)]]_α(Q)
  | ...
```

The first case corresponds to the lack of an actual inner subtyping constraint, *i.e.,* $\mathcal{L}' = \varnothing$. In this case, the constraint is simply collected from the type T (this means, for example, that `A ::= CHOICE {a B} (WITH COMPONENTS {...})` is equivalent to `A ::= CHOICE {a B}`).

The second case applies when there is only one PRESENT constraint applying to a component l' and when the CHOICE type restricted to l' is not recursive: $\Gamma \Vdash \textsc{Choice}\ \mathrm{F}_{\{l'\}}$, *e.g.,* `A ::= CHOICE {a A, b REAL} (WITH COMPONENTS {a PRESENT, ...})` is rejected because `A ::= CHOICE {a A}` is recursive. In this case, the collected constraint is the same as for the restricted type with no presence constraint, *e.g.,* `A ::= CHOICE {a REAL, b INTEGER}(WITH COMPONENTS {a (0.0) PRESENT,...})` is equivalent to `A ::= CHOICE {a REAL} (WITH COMPONENTS {a (0.0),...})`.

```
    | ...
    | ¬∃l' ∈ L'.K(l') ◁ (___, Some PRESENT)
      ∧ ∃l' ∈ L'.K(l') ◁ (σ, ___) →
        let  β and γ be fresh variables
        and  K̄ = K_{L'\{l'}}  and  T' = CHOICE F_{L'\{l'}}
        in [[T', WITH COMPONENTS (Partial, K̄)]]_γ(Q)
           ∧̈ α ≐̈ (l' : β) ∪̇ γ
           ∧̈ match σ with None → [[F(l')]]_β(Q)
                        | Some (C) → [[F(l'), C]]_β(Q)
  end (* cases *)
| ...
```

The last case applies when there is no PRESENT constraint. A component labelled l' is chosen and the constraint corresponding to the CHOICE type without l' is collected:

$$[\![\mathrm{T}', \textsc{With components}\,(\mathsf{Partial}, \overline{\mathrm{K}})]\!]_\gamma(\mathcal{Q}).$$

If there is actually a subtyping constraint C for the component l', then the constraint for it is collected: $[\![\mathrm{F}(l'), \mathrm{C}]\!]_\beta(\mathcal{Q})$. Otherwise the constraint from the component alone is collected: $[\![\mathrm{F}(l')]\!]_\beta(\mathcal{Q})$. The next cases correspond to the SET and SEQUENCE types:

$$\begin{array}{l}
\mid \ldots \\
\mid (\textsc{Set}\,\Phi_{\mathcal{L}} \mid \textsc{Sequence}\,\Phi_{\mathcal{L}})\ \textsf{when}\ \mathcal{L}' \subseteq \mathcal{L} \rightarrow \\
\quad \textsf{cases}\ \mathcal{L}' = \varnothing \rightarrow [\![\mathrm{T}]\!]_\alpha(\mathcal{Q}) \\
\quad\quad \mid \exists l' \in \mathcal{L}'.\,(\mathrm{K}(l') \lhd (\sigma, \mathsf{Some}\,\textsc{Optional}) \\
\quad\quad\quad\quad\quad \wedge\ \Phi(l') \lhd (\mathrm{T}_0, \mathsf{Some}\,\textsc{Optional})) \rightarrow \\
\quad\quad\quad \textsf{let}\ \overline{\mathrm{K}}_{\mathcal{L}'} = \mathrm{K}_{\mathcal{L}'} \uplus \{l' \mapsto (\sigma, \mathsf{None})\} \\
\quad\quad\quad \textsf{in}\ [\![\mathrm{T}, \textsc{With components}\ (\mathsf{Partial}, \overline{\mathrm{K}}_{\mathcal{L}'})]\!]_\alpha(\mathcal{Q}) \\
\quad\quad \mid \ldots
\end{array}$$

The first pattern corresponds to the situation where there is an OPTIONAL constraint applying to a component marked as OPTIONAL in the type definition. Hence the presence constraint is removed, *e.g.,* `A ::= SET {a REAL OPTIONAL} (WITH COMPONENTS {a (0.0) OPTIONAL, ...})` is equivalent to `A ::= SET {a REAL OPTIONAL} (WITH COMPONENTS {a (0.0) ...})`.

$$\begin{array}{l}
\quad \mid \ldots \\
\quad \mid \exists l' \in \mathcal{L}'.\,(\mathrm{K}(l') \lhd (\mathsf{None}, \mathsf{Some}\,\textsc{Absent}) \\
\quad\quad\quad\quad\quad \wedge\ \Phi(l') \lhd (\mathrm{T}_0, \mathsf{Some}\,\textsc{Optional})) \rightarrow \\
\quad\quad \textsf{let}\ \mathrm{C}' = \textsc{With components}\ (\mathsf{Partial}, \mathrm{K}_{\mathcal{L}' \setminus \{l'\}}) \\
\quad\quad \textsf{in}\ \ [\![\textsc{Set}\ \Phi_{\mathcal{L} \setminus \{l'\}}, \mathrm{C}']\!]_\alpha(\mathcal{Q}) \\
\quad \mid \ldots
\end{array}$$

The second pattern rules if an ABSENT constraint *alone* applies to a component marked as OPTIONAL. Then the component and the constraints are simply discarded, *e.g.,* `A ::= SET {a REAL OPTIONAL} (WITH COMPONENTS {a ABSENT,...})` is equivalent to `A ::= SET {}`. If the ABSENT constraint was associated to another kind of constraint, like a value constraint, then it is an error (in the implementation, this behaviour can be turned into a warning). For instance, the semantics of `A ::= SET {a REAL OPTIONAL} (WITH COMPONENTS {a (0.0) ABSENT,...})` is undefined.

$$
\begin{array}{l}
\mid \ldots \\
\mid \exists l' \in \mathcal{L}'.\mathrm{K}(l') \lhd (\sigma, \mathsf{Some}\,\textsc{Present}) \rightarrow \\
\quad \textbf{let}\ \overline{\mathrm{K}}_{\mathcal{L}'} = \mathrm{K}_{\mathcal{L}'} \uplus \{l' \mapsto (\sigma, \mathsf{None})\}\ \textbf{in} \\
\quad\ \ \textbf{let}\ \mathrm{C}' = \textsc{With components}\ (\mathsf{Partial}, \overline{\mathrm{K}}_{\mathcal{L}'}) \\
\quad \textbf{in}\ (\textbf{match}\ \Phi(l')\ \textbf{with} \\
\qquad\quad (\mathrm{T}_0, \mathsf{Some}\,\textsc{Optional}) \rightarrow \\
\qquad\quad\ \ \textbf{let}\ \Phi'_{\mathcal{L}} = \Phi_{\mathcal{L}} \uplus \{l' \mapsto (\mathrm{T}_0, \mathsf{None})\} \\
\qquad\quad\ \ \textbf{in}\ (\textbf{match}\ \Gamma \Vdash \textsc{Set}\ \Phi'_{\{l'\}}\ \textbf{with} \\
\qquad\quad\qquad \textbf{true} \rightarrow [\![\textsc{Set}\ \Phi'_{\mathcal{L}}, \mathrm{C}']\!]_\alpha(\mathcal{Q})) \\
\qquad\quad \mid\ \_\_ \quad \rightarrow [\![\textsc{Set}\ \Phi_{\mathcal{L}}, \mathrm{C}']\!]_\alpha(\mathcal{Q})) \\
\mid \ldots
\end{array}
$$

The third pattern is appropriate when there is some PRESENT constraint. If the component in question is not optional, then the presence constraint is discarded, otherwise its mark OPTIONAL is discarded and so is the constraint (we need to check also that the resulting type is still well-founded), *e.g.,* `A ::= SET {a A OPTIONAL} (WITH COMPONENTS {a PRESENT, ...})` is equivalent to `A ::= SET {a A}`, which has only infinite values, hence must be rejected. The last case is:

$$
\begin{array}{l}
\mid \ldots \\
\mid \exists l' \in \mathcal{L}'.\mathrm{K}(l') \lhd (\sigma, \mathsf{None}) \rightarrow \\
\quad \textbf{let}\ \ \beta\ \textit{and}\ \gamma\ \textit{be fresh variables} \\
\quad \textbf{and}\ \mathrm{C}' = \textsc{With components}\ (\mathsf{Partial}, \mathrm{K}_{\mathcal{L}' \setminus \{l'\}}) \\
\quad \textbf{in}\ [\![\textsc{Set}\ \Phi_{\mathcal{L}\setminus\{l'\}}, \mathrm{C}']\!]_\gamma(\mathcal{Q})\ \ddot{\wedge} \\
\qquad \textbf{match}\ \Phi(l')\ \textbf{with} \\
\qquad\quad (\mathrm{T}_0, \mathsf{None} \mid \mathsf{Some}\,(\textsc{Default}\ \_\_)) \rightarrow \\
\qquad\quad\ \ \alpha \doteq \text{`}\mathsf{Bind}\,(l', \beta, \gamma)\ \ddot{\wedge} \\
\qquad\quad\ \ (\textbf{match}\ \sigma\ \textbf{with}\ \mathsf{None} \rightarrow [\![\mathrm{T}_0]\!]_\beta(\mathcal{Q}) \\
\qquad\quad\qquad \mid \mathsf{Some}\,(\mathrm{C}_0) \rightarrow [\![\mathrm{T}_0, \mathrm{C}_0]\!]_\beta(\mathcal{Q})) \\
\qquad \mid (\mathrm{T}_0, \mathsf{Some}\,\textsc{Optional}) \rightarrow \\
\qquad\quad\ \ \alpha \doteq \text{`}\mathsf{Bind}\,(l', \beta, \gamma)\,\dot{\cup}\,\gamma\ \ \ddot{\wedge} \\
\qquad\quad\ \ \textbf{match}\ \sigma\ \textbf{with}\ \mathsf{None} \rightarrow [\![\mathrm{T}_0]\!]_\beta(\mathcal{Q}) \\
\qquad\quad\qquad \mid \mathsf{Some}\,(\mathrm{C}_0) \rightarrow [\![\mathrm{T}_0, \mathrm{C}_0]\!]_\beta(\mathcal{Q}) \\
\textbf{end}\ \ (*\ \textbf{cases}\ *)
\end{array}
$$

This last pattern applies when there is no presence constraint on the component (see None in $\mathrm{K}(l') \lhd (\sigma, \mathsf{None})$). A component labelled l' is chosen, and the constraint corresponding to T without l' is collected: $[\![\textsc{Set}\ \Phi_{\mathcal{L}\setminus\{l'\}}, \mathrm{C}']\!]_\gamma(\mathcal{Q})$ (if T is actually a SEQUENCE type, we nevertheless transform it in SET because their semantics is the same in our work).

Next, if there is actually a subtyping constraint C for the component l', then the constraint for this subtyped component is collected: $[\![\mathrm{T}_0, \mathrm{C}_0]\!]_\beta(\mathcal{Q})$. Otherwise the constraint from the component alone is collected: $[\![\mathrm{T}_0]\!]_\beta(\mathcal{Q})$. Finally, if the component l' is not OPTIONAL, then the constraint for T is collected: $\alpha \doteq \text{`}\mathsf{Bind}\,(l', \beta, \gamma)$. Otherwise, we have to take into account that the component may be missing in the value notation: $\alpha \doteq \text{`}\mathsf{Bind}\,(l', \beta, \gamma)\,\dot{\cup}\,\gamma$.

6.11 Type reference

$$
\begin{array}{l}
[\![\text{`}\mathsf{TRef}\,(x), \mathrm{C}]\!]_\alpha(\mathcal{Q}_\mathcal{R}) = \\
\quad \textbf{if } x \in \mathcal{R} \textbf{ then } \alpha \doteq \mathcal{Q}(x) \\
\quad \textbf{else match } \Gamma(x) \textbf{ with} \\
\qquad\quad (\mathrm{T}_0, \mathsf{None}) \rightarrow\ [\![\mathrm{T}_0, \mathrm{C}]\!]_\alpha(\mathcal{Q} \uplus \{x \mapsto \alpha\}) \\
\qquad | \ (\mathrm{T}_0, \mathsf{Some}\,\mathrm{C}_0) \rightarrow \\
\qquad\quad\ [\![\mathrm{T}_0, \mathrm{C}\ \textsc{Intersection}\ \mathrm{C}_0]\!]_\alpha(\mathcal{Q} \uplus \{x \mapsto \alpha\})
\end{array}
$$

This equation defines the constraint collection from type references *with a subtyping constraint* — here C. This situation can only happen when a component, which is always a reference in core ASN.1 (see section 6), is constrained by C. It is similar to the equation $[\![\text{`}\mathsf{TRef}\,(x)]\!]_\alpha(\mathcal{Q}_\mathcal{R})$ we gave in the section 5. As before, two cases can occur. First, if the reference x has an image through the mapping $\mathcal{Q}$, it means we previously analysed the referenced type, and we collect $\alpha \doteq \mathcal{Q}(x)$.

Otherwise, we analyse the referenced subtype $\Gamma(x)$. The first pattern corresponds to the case when it is not a proper subtype, *i.e.,* it is actually a type T_0: we collect the constraints from it *with the subtyping constraint* C: $[\![\mathrm{T}_0, \mathrm{C}]\!]_\alpha(\mathcal{Q} \uplus \{x \mapsto \alpha\})$. This case means that the values of `A` in `A ::= SET {a B} (WITH COMPONENTS {a (0.0)})`, where `B ::= REAL`, are the same as `A` in `A ::= SET {a B}`, where `B ::= REAL (0.0)`.

The second pattern applies when the referenced type T_0 is itself constrained by C_0. Then we form the constraint C Intersection C_0 and to apply it to T_0. This case means that the values of `A` in `A ::= SET {a B} (WITH COMPONENTS {a (0.0)})`, where `B ::= REAL (-2..15)`, are the same as `A` in `A ::= SET {a B}`, where `B ::= REAL ((0.0) INTERSECTION (-2..15))`.

In the following, we show the constraint collection from a complex subtype and the solved form. Let us consider the following specification excerpt:

```
T ::= SET (ALL EXCEPT
              ((SIZE (6..9))
               INTERSECTION
               (WITH COMPONENT (16..19))))
      OF INTEGER
```

We want the constraint whose solution in α is the set of terms of `T`, shown in Figure 2.

The simplified result of the solving procedure (not presented here), is:

$$
\begin{array}{ll}
& [\![\text{`TRef ("T")}]\!]_\alpha(\{\}) \\
= & \quad [\![\textsc{Set of Integer}]\!]_\beta(\{\}) \\
& \quad \ddot{\wedge}\ [\![\textsc{Set of Integer},\ \textsc{Size}\,(\text{`Interval}\,(\text{`PosInt}\,(6), \text{`PosInt}\,(9))) \\
& \qquad \textsc{Intersection}\,(\textsc{With component}\ (\text{`Interval}\,(\text{`PosInt}\,(16), \text{`PosInt}\,(19))))]\!]_\varepsilon(\{\}) \\
& \quad \ddot{\wedge}\ \alpha \doteq \beta \dot{\setminus} \varepsilon \\
= & \quad ([\![\textsc{Integer}]\!]_\gamma(\{\})\ \ddot{\wedge}\ \delta \doteq \text{`Cons}\,(\gamma, \delta) \dot{\cup} \text{`Nil} \\
& \qquad \ddot{\wedge}\ \beta \doteq \delta \diamond \mathbb{N}^+) \\
& \quad \ddot{\wedge}\ ([\![\textsc{Set of Integer}, \textsc{Size}\ (\text{`Interval}\,(\text{`PosInt}\,(6), \text{`PosInt}\,(9)))]\!]_\zeta(\{\}) \\
& \qquad \ddot{\wedge}\ [\![\textsc{Set of Integer}, \textsc{With component}\ (\text{`Interval}\,(\text{`PosInt}\,(16), \text{`PosInt}\,(19))))]\!]_\lambda(\{\}) \\
& \qquad \ddot{\wedge}\ \varepsilon \doteq \zeta \dot{\cap} \lambda) \\
& \quad \ddot{\wedge}\ \alpha \doteq \beta \dot{\setminus} \varepsilon \\
= & \quad \gamma \doteq \mathbb{N}\ \ddot{\wedge}\ \delta \doteq \text{`Cons}\,(\gamma, \delta) \dot{\cup} \text{`Nil}\ \ddot{\wedge}\ \beta \doteq \delta \diamond \mathbb{N}^+ \\
& \quad \ddot{\wedge}\ ([\![\textsc{Integer}]\!]_\theta(\{\})\ \ddot{\wedge}\ \eta \doteq \text{`Cons}\,(\theta, \eta) \dot{\cup} \text{`Nil} \\
& \qquad \ddot{\wedge}\ \zeta \doteq \eta \diamond \text{`Interval}\,(\text{`PosInt}\,(6), \text{`PosInt}\,(9))) \\
& \quad \ddot{\wedge}\ ([\![\textsc{Integer}, \text{`PosInt}(16) \leqslant .. \leqslant \text{`PosInt}(19)]\!]_\iota(\{\}) \\
& \qquad \ddot{\wedge}\ \xi \doteq \text{`Cons}\,(\iota, \xi) \dot{\cup} \text{`Nil}\ \ddot{\wedge}\ \lambda \doteq \xi \diamond \mathbb{N}^+) \\
& \quad \ddot{\wedge}\ \varepsilon \doteq \zeta \dot{\cap} \lambda\ \ddot{\wedge}\ \alpha \doteq \beta \dot{\setminus} \varepsilon \\
= & \quad \gamma \doteq \mathbb{N}\ \ddot{\wedge}\ \delta \doteq \text{`Cons}\,(\gamma, \delta) \dot{\cup} \text{`Nil}\ \ddot{\wedge}\ \beta \doteq \delta \diamond \mathbb{N}^+ \\
& \quad \ddot{\wedge}\ \theta \doteq \mathbb{N}\ \ddot{\wedge}\ \eta \doteq \text{`Cons}\,(\theta, \eta) \dot{\cup} \text{`Nil} \\
& \quad \ddot{\wedge}\ \zeta \doteq \eta \diamond \text{`Interval}\,(\text{`PosInt}\,(6), \text{`PosInt}\,(9)) \\
& \quad \ddot{\wedge}\ \iota \doteq \text{`Interval}\,(\text{`PosInt}\,(16), \text{`PosInt}\,(19)) \\
& \quad \ddot{\wedge}\ \xi \doteq \text{`Cons}\,(\iota, \xi) \dot{\cup} \text{`Nil}\ \ddot{\wedge}\ \lambda \doteq \xi \diamond \mathbb{N}^+ \\
& \quad \ddot{\wedge}\ \varepsilon \doteq \zeta \dot{\cap} \lambda\ \ddot{\wedge}\ \alpha \doteq \beta \dot{\setminus} \varepsilon
\end{array}
$$

Figure 2: Constraint collection from a complex subtype

$$
\left\{
\begin{array}{ll}
\alpha \doteq & \quad \omega \diamond \text{`Interval}\,(\text{`PosInt}\,(6), \text{`PosInt}\,(9)) \\
& \dot{\cup}\ \delta \diamond (\quad \text{`Interval}\,(\text{`PosInt}\,(0), \text{`PosInt}\,(5)) \\
& \qquad \dot{\cup}\ \text{`Interval}\,(\text{`PosInt}\,(10), \text{`PlusInfInt})) \\
\omega \doteq & \text{`Cons}\,(\phi, \nu) \\
\phi \doteq & \quad \text{`Interval}\,(\text{`MinInfInt}, \text{`PosInt}\,(15)) \\
& \dot{\cup}\ \text{`Interval}\,(\text{`PosInt}\,(20), \text{`PlusInfInt}) \\
\nu \doteq & \text{`Cons}\,(\phi, \nu) \dot{\cup} \text{`Nil} \\
\delta \doteq & \text{`Cons}\,(\mathbb{N}, \delta) \dot{\cup} \text{`Nil}
\end{array}
\right.
$$

So, the values of type T are either of size $[6; 9]$ and made of integers in $]-\infty; 15] \cup [20; +\infty[$, or of size $[0; 5] \cup [10; +\infty[$ and made of integers in $\mathbb{N}$. As a corollary, this type contains at least a finite value, hence is correct.

7 Full Collection and Solving

This section completes the two previous sections which present the constraint collection from types and subtypes and it describes the solving procedure for the collected constraints. As a result, either the constraints have no solutions (and the corresponding ASN.1 specification must be rejected) or the value sets can be finitely represented. It is straightforward to determine whether these value sets are empty; if they are empty then the specification is rejected.

The constraint modeling an ASN.1 specification is

$$\overline{\kappa} = \ddot{\bigwedge_{x \in \mathcal{X}}} [\![\text{`TRef}\,(x)]\!]_{\mathcal{Q}(x)}(\mathcal{Q}_{\mathcal{X} \setminus \{x\}})$$

where $\mathcal{X} \subset \mathcal{R}$ is the finite set of the top-level type names of the specification and $\mathcal{Q} : \mathcal{X} \to V$ maps these type names to fresh variables. The construction of a constraint modeling an entire specification is convenient during the collection (less parenthesis involved), but it is more comfortable for the solving procedure to use a system of constraints.

Definition 7.1 (Systems of constraints)**.** *A* system of constraints *is simply a set of constraints. The system corresponding to the ASN.1 specification is* $\Xi(\overline{\kappa})$*, where*

$$\begin{array}{rcl} \Xi(\kappa_0 \mathbin{\ddot{\wedge}} \kappa_1) & = & \Xi(\kappa_0) \cup \Xi(\kappa_1) \\ \Xi(\kappa) & = & \{\kappa\} \textit{ otherwise.} \end{array}$$

We introduce the solving procedure as the semantics of a system of constraints. In this aim, we first need to define a strict subset of the expressions E containing no variables: the *terms*. They can be characterised by the variables occurring in the expressions:

Definition 7.2 (Variables in expressions)**.** *The countable set of variables* α*,* β*,* γ *etc. is noted* V*. The variables of an expression* e *are given by:*

$$\begin{array}{rcl} \nu(\alpha) & = & \{\alpha\} \\ \nu(\dot{\neg} e) & = & \nu(e) \\ \nu(e_0 \mathbin{\dot{\cup}} e_1) & = & \nu(e_0) \cup \nu(e_1) \\ \nu(e_0 \mathbin{\dot{\cap}} e_1) & = & \nu(e_0) \cup \nu(e_1) \\ \nu(e_0 \mathbin{\dot{\setminus}} e_1) & = & \nu(e_0) \cup \nu(e_1) \\ \nu(\text{`Cons}\,(e_0, e_1)) & = & \nu(e_0) \cup \nu(e_1) \\ \nu(\text{`Bind}\,(l, e_0, e_1)) & = & \nu(e_0) \cup \nu(e_1) \\ \nu(l : e) & = & \nu(e) \\ \nu(e \diamond \varsigma) & = & \nu(e) \\ \nu(e) & = & \varnothing \textit{ otherwise.} \end{array}$$

Now we can define the ground expressions, or terms.

Definition 7.3 (Terms)**.** *A* term t *is an element of the set* $\mathcal{H} \triangleq \{e \in \mathrm{E} \mid \nu(e) = \varnothing\}$*. The set of all subsets of* $\mathcal{H}$ *is noted* $\wp(\mathcal{H})$*.*

Note that the terms do not only correspond to the ASN.1 well-typed values, for example the term $\text{`Cons}\,(\text{`PosInt}\,(0), \text{`Cons}\,(\text{String ""}, \text{`Nil}))$ is not a typable term, i.e. there is no ASN.1 type for the ASN.1 value `{0, ""}` denoted by this term.

Another necessary step is to characterise different kind of expressions, because the constraints in which they appear are solved by means of a specific algorithm: one for the integer intervals, one for the real intervals, one for the regular expressions, one for the powersets and one for the sets.

Definition 7.4 (Characterization of integer intervals)**.** *An expression* $e \in \mathrm{E}$ *denotes an integer interval if* $\chi_{\mathrm{I}}\,(e) = \textbf{true}$*, where*

$$\begin{array}{l}
\textbf{let rec}\ \chi_{\mathrm{I}} = \textbf{function}\ \dot{\neg} e \to \chi_{\mathrm{I}}(e) \\
\quad \mid e_0 \dot{\cup} e_1 \mid e_0 \dot{\cap} e_1 \mid e_0 \dot{\setminus} e_1 \to \chi_{\mathrm{I}}(e_0)\ \&\&\ \chi_{\mathrm{I}}(e_1) \\
\quad \mid \alpha\ \textbf{when}\ \exists e \in \mathrm{E}.(\alpha \ddot{=} e) \in \Xi\,(\overline{\kappa}) \to \chi_{\mathrm{I}}(e) \\
\quad \mid \dot{0} \mid \text{`Interval}\ __ \to \textbf{true} \mid __ \to \textbf{false}
\end{array}$$

Definition 7.5 (Characterization of real intervals)**.** *An expression* $e \in \mathrm{E}$ *denotes a real interval if* $\chi_{\mathrm{F}}\,(e) = \textbf{true}$*, where*

$$\begin{array}{l}
\textbf{let rec}\ \chi_{\mathrm{F}} = \textbf{function}\ \dot{\neg} e \to \chi_{\mathrm{F}}(e) \\
\quad \mid e_0 \dot{\cup} e_1 \mid e_0 \dot{\cap} e_1 \mid e_0 \dot{\setminus} e_1 \to \chi_{\mathrm{F}}(e_0)\ \&\&\ \chi_{\mathrm{F}}(e_1) \\
\quad \mid \alpha\ \textbf{when}\ \exists e \in \mathrm{E}.(\alpha \ddot{=} e) \in \Xi\,(\overline{\kappa}) \to \chi_{\mathrm{F}}(e) \\
\quad \mid \dot{0} \mid __\ __\ ..\ __\ __ \to \textbf{true} \mid __ \to \textbf{false}
\end{array}$$

Definition 7.6 (Characterization of regular expressions)**.** *An expression* $e \in \mathrm{E}$ *denotes a regular expression if* $\chi_{\mathrm{R}}\,(e) = \textbf{true}$*, where*

$$\begin{array}{l}
\textbf{let rec}\ \chi_{\mathrm{R}} = \textbf{function}\ \dot{\neg} e \to \chi_{\mathrm{R}}(e) \\
\quad \mid e_0 \dot{\cup} e_1 \mid e_0 \dot{\cap} e_1 \mid e_0 \dot{\setminus} e_1 \to \chi_{\mathrm{R}}(e_0)\ \&\&\ \chi_{\mathrm{R}}(e_1) \\
\quad \mid \alpha\ \textbf{when}\ \exists e \in \mathrm{E}.(\alpha \ddot{=} e) \in \Xi\,(\overline{\kappa}) \to \chi_{\mathrm{R}}(e) \\
\quad \mid \text{`Regexp}\ __ \to \textbf{true} \mid __ \to \textbf{false}
\end{array}$$

Definition 7.7 (Characterization of powerset expressions)**.** *An expression* $e \in \mathrm{E}$ *denotes a powerset if* $\chi_{\mathrm{P}}\,(e) = \textbf{true}$*, where*

$$\begin{array}{l}
\textbf{let rec}\ \chi_{\mathrm{P}} = \textbf{function}\ \dot{\neg} e \to \chi_{\mathrm{P}}(e) \\
\quad \mid e_0 \dot{\cup} e_1 \mid e_0 \dot{\cap} e_1 \mid e_0 \dot{\setminus} e_1 \to \chi_{\mathrm{P}}(e_0)\ \&\&\ \chi_{\mathrm{P}}(e_1) \\
\quad \mid \alpha\ \textbf{when}\ \exists e \in \mathrm{E}.(\alpha \ddot{=} e) \in \Xi\,(\overline{\kappa}) \to \chi_{\mathrm{P}}(e) \\
\quad \mid e \diamond \varsigma \to \textbf{true} \mid __ \to \textbf{false}
\end{array}$$

Definition 7.8 (Characterization of set expressions)**.** *An expression* $e \in \mathrm{E}$ *denotes a set if* $\chi_{\mathrm{S}}\,(\varnothing)\,(e) = \textbf{true}$*, where*

$$\begin{array}{l}
\textbf{let rec}\ \chi_{\mathrm{S}}\,(H) = \textbf{function}\ \dot{\neg} e \to \chi_{\mathrm{S}}\,(H)\,(e) \\
\quad \mid e_0 \dot{\cup} e_1 \mid e_0 \dot{\cap} e_1 \mid e_0 \dot{\setminus} e_1 \to \chi_{\mathrm{S}}\,(H)\,(e_0)\ \&\&\ \chi_{\mathrm{S}}\,(H)\,(e_1) \\
\quad \mid \alpha\ \textbf{when}\ \alpha \in H \to \textbf{true} \\
\quad \mid \alpha\ \textbf{when}\ \exists e \in \mathrm{E}.(\alpha \ddot{=} e) \in \Xi\,(\overline{\kappa}) \to \chi_{\mathrm{S}}\,(\{\alpha\} \cup H)\,(e) \\
\quad \mid \text{`Interval}\ \_\_ \mid \_\_\ \_\_\ ..\ \_\_\ \_\_ \mid \text{`Regexp}\ __ \mid e \diamond \varsigma \to \textbf{false} \\
\quad \mid __ \to \textbf{true}
\end{array}$$

Now we can define the semantics of expressions.

Definition 7.9 (Semantics of expressions)**.** *A* substitution σ *is a function* $\sigma : V \to \wp(\mathcal{H})$ *from variables to sets of terms. The set of substitutions is noted* Σ*. The standard semantics* $\dot{\mu}$ *of expressions is a mapping* $\dot{\mu} : \mathrm{E} \times \Sigma \to \wp(\mathcal{H})$ *from the Cartesian product of expressions and substitutions into sets of terms.*

- if $\chi_{\mathrm{I}}(e) = \textbf{true}$ then $\dot{\mu}(e, \sigma) = \|e\|$, where $\|\_\_\_\| : \mathrm{E} \to [>\mathit{closed\_int\_interval} \mid \dot{0}]$ is a function defined as follows:

$$\begin{array}{rl} \|e\| = & \textbf{let } \alpha \textit{ be a fresh variable} \\ & \textbf{in } \textit{solve\_integers}\,(\alpha \doteq e)\,(\alpha) \end{array}$$

- if $\chi_{\mathrm{F}}(e) = \textbf{true}$ then $\dot{\mu}(e, \sigma)$ is computed by a function similar to $\|\_\_\_\|$, which applies to real intervals (not presented for the sake of brevity);

- if $\chi_{\mathrm{R}}(e) = \textbf{true}$ then

$$\begin{array}{l} \dot{\mu}(e, \sigma) = \\ \quad \textbf{let } \alpha \textit{ be a fresh variable} \\ \quad \textbf{in match } \textit{solve\_regexp}\,(\alpha \doteq e)\,(\alpha) \textbf{ with} \\ \qquad s \textbf{ when } s \neq \texttt{""} \to \{\text{`Regexp}\,(s)\} \end{array}$$

- if $\chi_{\mathrm{P}}(e) = \textbf{true}$ then $\dot{\mu}$ is defined by the equations

$$\begin{array}{rcl} \dot{\mu}(\alpha, \sigma) & = & \sigma(\alpha) \\ \dot{\mu}(e\dot{\diamond}\varsigma, \sigma) & = & \{t\dot{\diamond}\varsigma \mid t \in \dot{\mu}(e, \sigma)\} \\ \dot{\mu}(\pi_0\dot{\cap}\pi_1, \sigma) & = & \textbf{let } \overline{\pi} = \dot{\mu}(\pi_0, \sigma), \dot{\mu}(\pi_1, \sigma) \\ & & \textbf{in } \varphi(\overline{\pi}, \dot{\cap}, \dot{\cap}) \\ \dot{\mu}(\pi_0\dot{\setminus}\pi_1, \sigma) & = & \textbf{let } \overline{\pi} = \dot{\mu}(\pi_0, \sigma), \dot{\mu}(\pi_1, \sigma) \\ & & \textbf{in } \varphi(\overline{\pi}, \dot{\cap}, \dot{\setminus}) \sqcup \varphi(\overline{\pi}, \dot{\setminus}, \mathit{fst}) \\ \dot{\mu}(\pi_0\dot{\cup}\pi_1, \sigma) & = & \\ \multicolumn{3}{l}{\quad \textbf{let } (\overline{\pi}_0, \overline{\pi}_1) \textbf{ as } \overline{\pi} = \dot{\mu}(\pi_0, \sigma), \dot{\mu}(\pi_1, \sigma)} \\ \multicolumn{3}{l}{\quad \textbf{in } \varphi(\overline{\pi}, \dot{\cap}, \dot{\cup}) \sqcup \varphi(\overline{\pi}, \dot{\setminus}, \mathit{fst}) \sqcup \varphi((\overline{\pi}_1, \overline{\pi}_0), \dot{\setminus}, \mathit{fst})} \end{array}$$

The second equation relies on the semantics of set expressions presented in definition 7.9 (e ranges over the set expressions). The second equation gives the semantics of the intersection of two powersets. $\overline{\pi}$ is the pair of the semantics of π_0 and π_1. The function φ is defined as follows:

$$\begin{array}{l} \varphi((\overline{\pi}_0, \overline{\pi}_1), f, g) \triangleq \{e\dot{\diamond}i \mid e_0\dot{\diamond}i_0 \in \overline{\pi}_0, e_1\dot{\diamond}i_1 \in \overline{\pi}_1, \\ \qquad\qquad e \in \|g(e_0, e_1)\| \setminus \{\dot{0}\}, i \in \|f(i_0, i_1)\| \setminus \{\dot{0}\}\} \end{array}$$

The first argument of φ is the pair $(\overline{\pi}_0, \overline{\pi}_1)$ of semantics. The second, f, is the set operator to be applied to the intervals of the elements of $\overline{\pi}_0$ and $\overline{\pi}_1$. The last, g, is the set operator to be applied to the set expressions of $\overline{\pi}_0$ and $\overline{\pi}_1$. For instance, $\varphi(\overline{\pi}, \dot{\cap}, \dot{\cap}) = \{e\dot{\diamond}i \mid e_0\dot{\diamond}i_0 \in \overline{\pi}_0, e_1\dot{\diamond}i_1 \in \overline{\pi}_1, e \in \|e_0 \dot{\cap} e_1\| \setminus \{\dot{0}\}, i \in \|i_0 \dot{\cap} i_1\| \setminus \{\dot{0}\}\}$. The intuition is that the intersection of two powersets is the powerset whose elements are

the intersection of the initial elements with the *same* cardinal (hence we compute the intersection of the intervals).

The fourth equation defines the semantics of the difference of two powersets: $\varphi\,(\overline{\pi}, \dot{\cap}, \dot{\setminus}) \sqcup \varphi\,(\overline{\pi}, \dot{\setminus}, \mathit{fst})$. The symbol $\sqcup$ is the disjunctive set union $\cup$. The elements of the powerset $\varphi\,(\overline{\pi}, \dot{\cap}, \dot{\setminus})$ are the difference between the initial elements of same cardinal. The elements of the powerset $\varphi\,(\overline{\pi}, \dot{\setminus}, \mathit{fst})$ are the elements of $\overline{\pi}_0$ whose cardinals are different from the cardinals of the elements of $\overline{\pi}_1$. The rationale of the equation comes from the trivial formula $A = (A \cap B) \sqcup (A \setminus B)$.

The last equation defines the semantics of the union of two powersets as $\varphi\,(\overline{\pi}, \dot{\cap}, \dot{\cup}) \sqcup \varphi\,(\overline{\pi}, \dot{\setminus}, \mathit{fst}) \sqcup \varphi\,((\overline{\pi}_1, \overline{\pi}_0), \dot{\setminus}, \mathit{fst})$. First, the elements of the powerset $\varphi\,(\overline{\pi}, \dot{\cap}, \dot{\cup})$ are the union of the initial elements of same cardinal. Second, the elements of the powerset $\varphi\,(\overline{\pi}, \dot{\setminus}, \mathit{fst})$ are the elements of $\overline{\pi}_0$ whose cardinals are different from the cardinals of the elements of $\overline{\pi}_1$. Last, the elements of the powerset $\varphi\,((\overline{\pi}_1, \overline{\pi}_0), \dot{\setminus}, \mathit{fst})$ are the elements of $\overline{\pi}_1$ whose cardinals are different from the cardinals of the elements of $\overline{\pi}_0$. The rationale of this equation is the formula: $A \cup B = (A \cap B) \sqcup (A \setminus B) \sqcup (B \setminus A)$.

- Otherwise, if $\chi_{\mathrm{S}}\,(\varnothing)\,(e) = \textbf{true}$ then $\dot{\mu}$ is defined by the following equations:

$$
\begin{array}{rcl}
\dot{\mu}(\dot{\mathsf{0}}, \sigma) & = & \varnothing \\
\dot{\mu}(\dot{\mathsf{1}}, \sigma) & = & \mathcal{H} \\
\dot{\mu}(\alpha, \sigma) & = & \sigma\,(\alpha) \\
\dot{\mu}(e_0 \,\dot{\cup}\, e_1, \sigma) & = & \dot{\mu}(e_0, \sigma) \cup \dot{\mu}(e_1, \sigma) \\
\dot{\mu}(e_0 \,\dot{\cap}\, e_1, \sigma) & = & \dot{\mu}(e_0, \sigma) \cap \dot{\mu}(e_1, \sigma) \\
\dot{\mu}(\dot{\neg} e, \sigma) & = & \mathcal{H} \setminus \dot{\mu}(e, \sigma) \\
\dot{\mu}(e_0 \,\dot{\setminus}\, e_1, \sigma) & = & \dot{\mu}(e_0 \,\dot{\cap}\, \dot{\neg} e_1, \sigma) \\
\dot{\mu}(l : e, \sigma) & = & \{l : t \mid t \in \dot{\mu}(e, \sigma)\} \\
\dot{\mu}(\text{`}\mathsf{Cons}\,(e_0, e_1), \sigma) & = & \\
\multicolumn{3}{r}{\{\text{`}\mathsf{Cons}\,(t_0, t_1) \mid t_0 \in \dot{\mu}(e_0, \sigma), t_1 \in \dot{\mu}(e_1, \sigma)\}} \\
\dot{\mu}(\text{`}\mathsf{Bind}\,(y, e_0, e_1), \sigma) & = & \\
\multicolumn{3}{r}{\{\text{`}\mathsf{Bind}\,(y, t_0, t_1) \mid t_0 \in \dot{\mu}(e_0, \sigma), t_1 \in \dot{\mu}(e_1, \sigma)\}} \\
\dot{\mu}(e, \sigma) & = & \{e\} \text{ otherwise.}
\end{array}
$$

Now we can define the semantics of constraints on top of the semantics of expressions.

Definition 7.10 (Semantics of constraints)**.** *We define the semantics $\ddot{\mu} : \mathcal{K} \times \Sigma \to \mathit{bool}$ of constraints as a predicate on the Cartesian product of constraints and substitutions:*

$$
\begin{array}{rcl}
\ddot{\mu}(\kappa_0 \,\ddot{\wedge}\, \kappa_1, \sigma) & = & \ddot{\mu}(\kappa_0, \sigma) \wedge \ddot{\mu}(\kappa_1, \sigma) \\
\ddot{\mu}(e_0 \,\ddot{\subseteq}\, e_1, \sigma) & = & \dot{\mu}(e_0, \sigma) \subseteq \dot{\mu}(e_1, \sigma) \\
\ddot{\mu}(e_0 \,\ddot{=}\, e_1, \sigma) & = & \ddot{\mu}((e_0 \,\ddot{\subseteq}\, e_1) \,\ddot{\wedge}\, (e_1 \,\ddot{\subseteq}\, e_0), \sigma)
\end{array}
$$

where $\wedge$ is the logical boolean operator and $\subseteq$ is the inclusion over mathematical sets (of terms).

Definition 7.11 (Solutions)**.** *The* set of solutions $\mathcal{S}(\kappa)$ *of a constraint* κ *is the set of all substitutions that satisfy the semantics of* κ*:* $\mathcal{S}(\kappa) = \{\sigma \in \Sigma \mid \ddot{\mu}(\kappa, \sigma)\}$.

The algorithm constructing the solutions of a system of set constraints may be the algorithm which was published by Aiken and Wimmers in 1992 [12]. We constructed our constraint expressions in order to almost exactly fit the input of this algorithm (see definition 4.1). The only thing to do is to replace $\mathrm{E}_0 \dot{\setminus} \mathrm{E}_1$ by $\mathrm{E}_0 \dot{\cap} \dot{\neg} \mathrm{E}_1$, since these two set expressions have the same semantics (see definition 7.9).

7.1 Worst-case complexity analysis

Let us say some words about the complexity of the solving procedure. The collecting algorithm we have presented is obviously not optimised, in any way. For instance, a type which is constrained by a combination of subtyping constraints will be analysed each time a basic constraint is analysed. The reason for this is that we want to provide an algorithm that can also be considered as a reference formal model, thus it must be as readable as possible. Therefore, any optimisation is postponed until the implementation phase (as using hash-consing or caches to solve the mentioned problem). Anyway, the reader can convince himself that the worst case complexity of the collecting algorithm is proportional to the size of the subtypes plus the complexity of the computations on integer intervals and regular expressions.

The worst case complexity of the solving procedure is the same as the complexity of Aiken and Wimmers' algorithm. In their paper, they prove that their algorithm is in class NEXPTIME (they also published, together with Kozen and Vardi, a general study about the complexity of solving set constraints [15]). This result may be very disappointing, but it is inherent to the high expressiveness of ASN.1 subtyping, in particular the use of a complement operator (hence not monotonic) with indiscriminate unions and intersections. As a future work, it would be interesting to implement our algorithm and to make some benchmarks. Also, there exists constraint subclasses with polynomial complexity in the worst case and it would be worth studying whether it is possible, in practice, to model ASN.1 with them.

Conclusion

It is a common belief among the practitioners in the networking area that theoretical study cannot help the industrial audience. We nevertheless think that many tools the engineers use can concretely benefit from such an initiative, especially those based on protocol languages and compilers. The work on ASN.1 we have presented in this article follows this track, and can be considered in several complementary ways.

From a theoretical point of view, we bring to the fore the first complete and formal semantics of X.680, based on the set constraints

framework. Each syntactic construct is mapped to a mathematical object in a consistent manner, thus bringing new insights to obscure areas of the standard, like type compatibility in assignments, type emptiness, type recursion in the presence of the complement subtyping constraint, etc. The set constraints prove to be a suitable model that allows one to encode all X.680. One of the reasons is that the paradigm of ASN.1 types comes from the telecommunications industry, where what matters are the values exchanged between applications or network equipments — hence types are naturally considered as *sets* of values. The tree-like structure of the ASN.1 values also contributed to this successful fitting. Nevertheless, some special values not found in the set constraint litterature, such as regular expressions, numeric intervals and powersets, motivated a specific contribution. Also, the usage of a functional meta-language to express our algorithm brought elegance and compactness. This work can be considered as an improvement of our PhD [16], which introduced an *ad-hoc* 'operational semantics' of X.680, based on the 1988/1990 standard — as opposed to the present 'denotational semantics' based on the 1997/2001 standard. Indeed, this early framework did not take advantage of the expressiveness of the set constraints, and their natural fitting to the 'types as sets' paradigm of ASN.1. Also, the operational aspect introduced too many aspects that are split here into collection and solving. From a general standpoint, a formal semantics, if properly documented, is a good complement to that provided by the standards. They indeed provide definitions in English by following the ASN.1 grammar in Backus-Naur Form. Unfortunately, this is ambiguous in the sense that a specification can be derived using different syntactical rules and, therefore, such a syntax-driven approach of semantics may be misleading.

From a practical point of view, the existence of a compiler for a plainer version of this meta-language, namely the OCaml language, helps in bridging the gap between the semantics and its implementation in an ASN.1 compiler. We identified the main issues in the validation of X.680 specifications, and stressed the importance of the 'at least one finite value' property for types in the Protocol Data Units. We showed how our semantics *is* naturally an algorithm.

Despite the present paper focusing on the telecommunication usages of ASN.1, we do not forget the database researchers. They are, in general, more concerned about formal foundations, hence we hope this work will convince them of the suitability of ASN.1 for their purposes. For instance, ongoing joint work of the ISO and the ITU-T explores the links between XML and ASN.1, in particular the use of ASN.1 as a schema notation for XML, and a mapping from XML schemas to ASN.1 modules (`http://asn1.elibel.tm.fr/xml/`).

Immediate further work ranges from optimization of the collecting algorithm, to extension to full ASN.1 (X.681, X.682 and X.683). A mid-term project would be to implement our algorithm in OCaml. This will lead to the completion of our parser and typechecker, Asno (available at `http://cristal.inria.fr/~rinderkn/`), which lacks subtype analysis. In the long term, a more challenging endeavour consists in

taking the benefit of our new semantics in order to formally model the Packed Encoding Rules [6].

References

[1] ITU-T Rec. X.680 (2002) or ISO/IEC 8824-1:2002. *Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation.*

[2] ITU-T Rec. X.681 (2002) or ISO/IEC 8824-2:2002. *Information technology – Abstract Syntax Notation One (ASN.1): Information object specification.*

[3] ITU-T Rec. X.682 (2002) or ISO/IEC 8824-3:2002. *Information technology – Abstract Syntax Notation One (ASN.1): Constraint specification.*

[4] ITU-T Rec. X.683 (2002) or ISO/IEC 8824-4:2002. *Information technology – Abstract Syntax Notation One (ASN.1): Parametrization of ASN.1 specifications.*

[5] ITU-T Rec. X.690 (2002) or ISO/IEC 8825-1:2002. *Information technology – ASN.1 Encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER).*

[6] ITU-T Rec. X.691 (2002) or ISO/IEC 8825-2:2002. *Information technology – ASN.1 Encoding Rules: Specification of Packed Encoding Rules (PER).*

[7] Olivier Dubuisson. *ASN.1 – Communication Between Heterogeneous Systems.* Academic Press, 2000. ISBN 0-12-6333361-0.

[8] John Larmouth. *ASN.1 Complete.* Morgan Kaufman, 2000. ISBN 0-12-233435-3.

[9] Masood Mortazavi. Sets and types: A review, June 1996.

[10] Alexander Aiken. Set Constraints: Results, Applications, and Future Directions. In *Second Workshop on the Principles and Practice of Constraint Programming*, volume 874 of *Lecture Notes on Computer Science*, pages 326–335, Orcas Island, Washington, May 1994. Springer Verlag.

[11] Leszek Pacholski and Andreas Podelski. Set Constraints: A Pearl in Research on Constraints. In *Third Conference on Principles and Practice of Constraint Programming*, volume 1330 of *Lecture Notes in Computer Science*, pages 549–562. Springer Verlag, 1997.

[12] Alexander Aiken and Edward Wimmers. Solving Systems of Set Constraints. In *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 329–340, Santa Cruz, California, June 1992. Extended Abstract.

[13] Guy Cousineau and Michel Mauny. *The Functional Approach to Programming.* Cambridge University Press, 1998. ISBN 0-521-57183-9 (hardcover), 0-521-57681-4 (paperback).

[14] Gérard Huet. Transducers as lexicon morphisms, phonemic segmentation by euphony analysis, application to a sanskrit tagger, 2002.

[15] Alexander Aiken, Dexter Kozen, Moshe Vardi, and Edward Wimmers. The Complexity of Set Constraints. In *Computer Science Logic*, volume 832 of *Lecture Notes in Computer Science*, pages 1–17, Swansea, Wales, September 1993. Springer Verlag.

[16] Christian Rinderknecht. *Une formalisation d'ASN.1 — Application d'une méthode formelle à un langage de spécification télécom.* PhD thesis, Université Pierre et Marie Curie (Paris 6), December 1998.