Formal Network Packet Processing with Minimal Fuss* †

Invertible Syntax Descriptions at Work

Reynald Affeldt†  David Nowak‡  Yutaka Oiwa‡

†Research Center for Information Security  ‡Information Technology Research Institute
National Institute of Advanced Industrial Science and Technology, Japan
{reynald.affeldt,david.nowak,y.oiwa}@aist.go.jp

Abstract
An error in an Internet protocol or its implementation is rarely benign: at best, it leads to malfunctions, at worst, to security holes. These errors are all the more likely that the official documentation for Internet protocols (the RFCs) is written in natural language. To prevent ambiguities and pave the way to formal verification of Internet protocols and their implementations, we advocate formalization of RFCs in a proof-assistant. As a first step towards this goal, we propose in this paper to use invertible syntax descriptions to formalize network packet processing. Invertible syntax descriptions consist in a library of combinators that can be used interchangeably as parsers or pretty-printers: network packet processing specified this way is not only unambiguous, it can also be turned into a trustful reference implementation, all the more trustful that there is no risk for inconsistencies between the parser and the pretty-printer. Concretely, we formalize invertible syntax descriptions in the Coq proof-assistant and extend them to deal with data-dependent constraints, an essential feature when it comes to parsing network packets. The usefulness of our formalization is demonstrated with an application to TLS, the protocol on which e-commerce relies.

Categories and Subject Descriptors D.3.2 [Programming Languages]: Language Classifications—Applicative (functional) languages; C.2.2 [Computer-Communication Networks]: Network Protocols—Protocol verification

General Terms Languages, Standardization, Verification

Keywords data-dependent grammar, Internet protocol, Coq, type classes, parsing and pretty-printing

1. Introduction

The official documentation for Internet protocols takes the form of memorandums published by the Internet Engineering Task Force: the so-called Requests for Comments (RFCs). They are the de facto standards for the development of network applications but since they are written in natural language, developers are sometimes led to resolve inherent ambiguities by reading the source code of other existing implementations. This “deciphering” task is all the more difficult that the specifications of Internet protocols are long and complex. For example, the specification of the Transport Layer Security protocol (TLS), the Internet protocol that provides privacy and data integrity to most e-commerce applications, consists of 104 pages [4], and is not self-contained. At best, errors in TLS implementations can disrupt the usage of network applications; at worst, they can be exploited by malicious users. In order to avoid these problems, we advocate the use of formal specification for Internet protocols. A formal specification not only eliminates ambiguities, but it also paves the way to formal verification of the protocol and its implementation.

As a first step towards formal specification of Internet protocols, we propose to use invertible syntax descriptions [18] to formalize network packet processing. Invertible syntax descriptions consist of a set of combinators that can be used interchangeably as parsers or pretty-printers (hereafter, printers). By formalizing network packet processing with invertible syntax descriptions, we actually fulfill two important objectives simultaneously: (1) we obtain an unambiguous grammar for the syntax of network packets, and (2) thanks to combinators, this formalization can be turned into reference implementations for both parsing and printing. Moreover, the reference implementation obtained in this way is arguably a trustful one because, and this is to put to the credit of invertible syntax descriptions, we get rid of inconsistencies that may arise from the parser and printer being developed separately.

However, invertible syntax descriptions as presented in [18] suffer several limitations. First, because they are implemented in the Haskell programming language [6], they do not provide the correctness guarantees expected of a trustful formalization. For example, the relations between parsers and printers, such as invertibility, cannot be checked formally. While formalizing invertible syntax descriptions in the Coq proof-assistant [21], we also provide formal proofs of properties (cf. Sect. 3.3) about parsing and printing, thus improving confidence in the formalization of network packet processing.

The second limitation of invertible syntax descriptions as presented in [18] is that they do not deal with data-dependent parsing. Given the advances in parsing technology, the untrained developer may be misled to think that getting a workable parser is just a matter of laying down the grammar into the appropriate tool. This is far from true, as experienced, for example, by Eric Rescorla, a recognized expert in TLS, who “attempted to write a grammar-driven parser (using Yacc) for the language with the aim of mechanically generating a decoder, but abandoned the project in frustration” [19].
The main difficulty is that the grammar of TLS packets is context-sensitive. For a concrete example, Sect. 7.4 of [4] defines Handshake packets as follows:

```plaintext
struct {
  HandshakeType msg_type;  /* handshake type */
  uint24 length;           /* bytes in message */
  select (HandshakeType) {
    case hello_request:     /* most cases omitted */
      HelloRequest;
    case finished:          /* some cases omitted */
      Finished;
  }
} body;
} Handshake;
```

The above structure is actually a dependent record. The `select` construct introduces a dependency between the last field `body` and the value of the first field `msg_type`. (This syntax is explained in Sect. 7.4 of [4]; we will come back to it in Sect. 4.1 of this paper). A clean way to specify such a context-sensitive syntax is to use data-dependent grammars, where the next grammar rule to apply might depend on the previously parsed value [11].

**Contributions** The purpose of this paper is to allow for specifying formally the syntax of binary protocol packets, and to automatically extract reference implementations of a parser and a printer. Our contributions are (1) a formalization in the Coq proof-assistant of invertible syntax descriptions that allow to unify parsing and printing, including formalization of properties such as invertibility, (2) an extension of invertible syntax descriptions to deal with data-dependent parsing (data-dependent constraint with w.r.t. the parsed value and also w.r.t. the input list of bytes), and (3) an application to the TLS protocol. The extraction feature of Coq then allows for generating automatically a certified parser and a certified printer.

**Outline** In Sect. 2, we briefly review the state of the art for parsing and printing, including invertible syntax descriptions. In Sect. 3, we introduce our formalization and extension of invertible syntax descriptions in the Coq proof-assistant. In Sect. 4, we investigate the use of invertible syntax descriptions for the specification of TLS network packets; we illustrate in particular data-dependent parsing and show that data-dependent constraints need not only be w.r.t. the parsed value but also w.r.t. the input list of bytes. In Sect. 5, we discuss practical considerations that come into play when dealing with a large formalization such as an RFC. In Sect. 6, we discuss work in progress, review related work in Sect. 7, and conclude in Sect. 8.

2. **Background on parsing and pretty-printing**

In this section, we briefly summarize and comment on the most common approaches to parsing and printing. Parsing (a.k.a. decoding or disassembling when dealing with network packets) consists in analyzing an input string (a sequence of bytes or characters) into its syntactic components according to a given grammar. It can either fail if the input string does not agree with the grammar, or return the syntactic components organized as an abstract syntax tree (AST). Printing (a.k.a. encoding or assembling) consists in the reverse process that transforms an AST into a string that it represents.

**Parsing with Yacc and its variants** In the programming languages community, it is common practice to use the parser generator Yacc [12] or one of its variants for implementing the parser in a compiler. Yacc takes as input the formal grammar of the programming language and output a parser for it. Although Yacc fits the needs for parsing simple programming languages, it has too much limitations for other uses such as the parsing of domain-specific languages, data formats, configuration files, networking protocols, etc. In fact, most of those parsers are done by hand and without using any formal grammar (see [11] for examples).

**Parsing with a monad** In functional programming, as an alternative to parser generator, it is common to encode a parser as a monad. A monad is an algebraic structure stemming from category theory [15] and that has proved useful to model imperative features in purely functional programming languages [23]. With this monadic approach, a parser of values of type `A` is encoded as a function that takes as input a list of bytes and either outputs the parsed value (of type `A`) and the remaining bytes, or fails in case of a syntax error [8]. This allows to define grammar constructions as combinators, i.e., higher-order functions that input parsers and output a new parser. Note that there were already parser combinators long before monads were introduced in functional programming (See for example [3]).

**Pretty-printing** As far as we know, for printing, there is no imperative programming tool that might be seen as the counterpart of Yacc. On the other hand, in functional programming there are combinator libraries to deal with printing (e.g., [7, 24]). Since they do not form a monad, it is not obvious how to unify them with combinator libraries for parsing. This means that we are still facing redundancy of and potential inconsistencies between two specifications of the same grammar.

**Invertible syntax descriptions** Rendel and Ostermann have recently proposed invertible syntax descriptions as a way to unify parsing and printing [18]. Invertible syntax descriptions is a unique programming interface that consists of combinators for describing a grammar. Those combinators are overloaded thanks to the type-class mechanism provided by the Haskell programming language. Overloading allows for giving terms completely different semantics depending on the context of their use. A typical example is the overloading of arithmetic operators. For example, in the expression `x + y` the semantics of `+` will be either integer or floating-point addition depending on the types of `x` and `y`. In invertible syntax descriptions, combinators are given two semantics: the first one as a parser and the second one as a printer.

3. **Data-dependent invertible syntax descriptions**

In this section, we propose a formalization of invertible syntax descriptions in Coq. It improves previous work [18] in two ways: (1) by formally delimiting cases when parsers and printers are indeed inverse of each other, and (2) by extending invertible syntax descriptions with combinators for data-dependent constraints [11]. Moreover, data-dependent constraints are not only w.r.t. parsed data but also w.r.t. the input list of bytes, which is important to formalize some kinds of network packets (see Sect. 4 for a concrete illustration using TLS).

In the following, first, we formalize partial isomorphisms (Sect. 3.1), a prerequisite for formalization of invertible syntax descriptions. Second, we formalize invertible syntax descriptions with data-dependent constraints, focusing on the most useful set of combinators for network packet processing (Sect. 3.2). Then, we comment on the formal relations between parsers and printers (Sect. 3.3).

3.1 Partial isomorphisms

A partial isomorphism is a pair of partial functions that are inverse of each other (on their domain) [18]. We formalize partial isomorphisms in Coq by the type `Iso`:

```plaintext
Record Iso (A B : Type) : Type := { apply : A → option B ;
  unapply : B → option A ;
  apply_unapply a b : apply a = Some b → unapply b = Some a ;
  unapply_apply a b : unapply b = Some a → apply a = Some b ;
};
```
Partial functions are formalized using the option type (fields apply and unapply). In order to prevent ill-defined partial isomorphisms, we require proofs that the partial functions are indeed invertible (fields apply_unapply and unapply_apply). From this definition we can deduce that, as expected of an inverse, it is unique when it exists. Note that this comes as an improvement over [18] because ill-defined partial isomorphisms cannot be prevented as easily in Haskell.

Program Definition cons_iso : 
Iso (A * list A) (list A) := {{ |
apply := λ al ⇒ Some (fst al :: snd al);
unapply := λ l ⇒
match l with nil ⇒ None | a::l' ⇒ Some (a,l') end | }}.

Program Definition cond_iso (cond:A → bool) :
Iso A A := {{ |
apply := λ a ⇒ if cond a then Some a else None;
unapply := λ a ⇒ if cond a then Some a else None | }}.

Program Definition undep_iso :
Iso (_:A & A) (A*B) := {{ |
apply := λ x ⇒ Some (proj1 x, proj2 x);
unapply := λ al ⇒ Some (existsT _ (fst len) (snd len)) | }}.

Program Definition chop_len_iso :
Iso (Z * list A) (list A) := {{ |
apply := λ nl ⇒ if Z_of_nat (length (snd nl)) == fst nl then Some (snd nl) else None;
unapply := λ a ⇒ Some (Z_of_nat (length l), l) | }}.

Program Definition Some_iso :
Iso (option A) (A) := {{ |
apply := λ a ⇒ Some a;
unapply := λ a ⇒ match a with |
Some a ⇒ Some a |
None ⇒ DEBUG "Some_iso" None end | }}.

Figure 1. Examples of partial isomorphisms

Fig. 1 provides examples of partial isomorphisms to be used later in this paper (Figures 3, 4, and 9). In one direction, cons_iso is a total function that adds a value at the head of a list. The inverse (partial) function splits a list into its head and its tail when it is non-empty, and fails otherwise. cond_iso is parametrized by a boolean condition: in both directions, it returns unchanged any input that satisfies the boolean condition and fails otherwise. undep_iso allows to see a non-dependent pair as a special case of dependent pair when the type of the second component does not depend on the value of the first component. chop_len_iso is the isomorphism between a pair made of a list and its length, and the list alone. Some_iso adds (and, in the other direction, removes) the constructor Some in front of a value (The function DEBUG used in the definition is explained in Sect. 5).

Note that only the definitions of the partial functions appear in Fig. 1: the proof obligations corresponding to apply_unapply and unapply_apply are generated by Coq and proved by the user (interactively in general, and automatically in the simplest cases).

3.2 Invertible syntax descriptions in Coq

We use the type class mechanism [20] of Coq to formalize invertible syntax descriptions. This takes the form of a class Syntax that defines the types of a set of combinators. Fig. 2 summarizes the most useful set of combinators for network packet processing. The class Syntax is parametrized by a type T for which we define two possible instances: one for parsing and one for printing.

The parser instance makes use of a parser whose type is the same as the one used in a parser monad (cf. Sect. 2):

Definition parser (A : Type) : Type := 
list byte → option (A * list byte).

Instance Syntax_parser : Syntax parser := {
(* see below for the instances of combinators *)
...].

A printer of values of type A is encoded as a function that takes as input a value of type A and either output a list of bytes that is the printing of the input value, or fail if the value is invalid w.r.t. the grammar:

Definition printer (A : Type) : Type := 
A → option (list byte).

Instance Syntax_printer : Syntax printer := {
(* see below for the instances of combinators *)
...].

As a result of this formalization, one has to write the grammar only once by using the combinators, and depending on the context of use it will either be interpreted as a parser or a printer for the encoded grammar rule.

Of course, combinators in the class Syntax must be meaningful as parsers as well as printers. We now go on explaining their instances.

3.2.1 Basic combinators (no data-dependent constraints)

The combinator Tok (for “token”), as a parser, returns the head of the input list of bytes, failing when the latter is empty (like in [18], we talk about tokens but the parser is in fact fed with bytes). In the parsing instance of the class Syntax, it is defined as follows:

Instance Syntax_parser : Syntax parser := 
Tok := λ s ⇒
match s with |
| nil ⇒ None |
}.1

1In order to make the parser a monad we would need an additional combinator of type T A → (A → T B) → T B. Although there would be an obvious choice for instantiating this combinator in the case of a parser, there is no meaningful choice in the case of a printer.
In order to parse dependently-typed data structures, we generalize the combinator `Prod` into a combinator `Prod_dep` that is dependently-typed and returns a parser or a printer for dependent pairs of types `(a : A & B a)`, where the type `B a` of the second component depends on the value of the first component `a` of type `A`. As a parser, `Prod_dep p1 p2` parses the first component of the dependent pair with `p1`, and then parses the second component with `p2 a` where `a` is the value parsed by `p1`: the parsing of the second component depends on the previously parsed value. It is similar to the `bind` combinator in a monadic parser except that `bind` only returns the second component. As a printer, `Prod_dep p1 p2` takes a dependent pair `(a, b)` as input and prints the first and second components with printers `p1` and `p2 a`, respectively.

As a concrete example, Fig. 4 displays a parser/printer for variable-length lists whose number of elements is encoded in a header. `repeat_dep p1 p2` is a parser (resp. a printer) for variable-length lists where `p1` is a parser (resp. printer) for positive integers used to parse (resp. print) the length and `p2` is a parser (resp. printer) for individual elements in the list.
Variables A B : Type.
Hypothesis eq_B_dec : EqDec B eq.

Program Definition proj_left_iso (b : B) :
  Iso (A * B) A := {}
  apply := \a \b \Rightarrow
    if (snd ab) == b then Some (fst ab) else None;
  unapply := \a \Rightarrow Some (a, b) \}.

Variable T : Type \to Type.
Hypothesis S : Syntax T.
Variable A : Type.

Program Definition sequential (p : parser A) (q : printer A) :
  Prop := \forall a, p s = Some (a, s) \to q a = Some s.

Lemma Prod_parser_printer :
  \forall (p : parser A) (q : printer A) : Prop.

Definition parser_printer {A : Type} :
  (p : parser A) (q : printer A) : Prop :=
  \forall s_1 s_2 a, p (s_1 ++ s_2) = Some (a, s_2) \to q a = Some s_1.

(ii) The property printer_parser states that if a value \( a \) is printed
into the list of bytes \( s_1 \) that is then parsed in a larger context
where \( s_2 \) is followed by a further list \( s_3 \), then it will be parsed
again as \( a \) with \( s_2 \) as the remaining list of bytes:

Definition printer_parser {A : Type} :
  (p : parser A) (q : printer A) : Prop :=
  \forall s_1 s_2 a, q a = Some s_1 \to p (s_1 ++ s_2) = Some (a, s_2).

We proved that parser_printer and printer_parser hold for
the combinators \( \text{Tok} \) and \( \text{Ret} \), and are preserved by the other combi-
nators of the class Syntax. For example, here follows the statement
for the product combinator:

Lemma Prod_parser_parser :
  \forall A B
  (p_1 : parser A)(p_2 : parser B)
  (q_1 : printer A)(q_2 : printer B),
  parser_parser p_1 q_1 \to parser_parser p_2 q_2 \to
  parser_parser (p_1 * p_2) (q_1 * q_2).

For proving that the relation parser_printer is preserved, we
need the additional hypothesis that the parser accesses its input
sequentially (it consumes bytes at the head of the input list and
do not modify the tail):

Definition sequential {A : Type} :
  (p : parser A) : Prop :=
  \forall s s_2 a, p s = Some (a, s_2) \to \exists s_1, s = s_1 ++ s_2.

This property holds for \( \text{Tok} \) and \( \text{Ret} \) and is preserved by other combinator.
We can thus, for example, prove that the product combinator
preserves the relation parser_printer assuming that sequential
holds for the parsers that are being paired:

Lemma Prod_parser_parser :
  \forall A B
  (p_1 : parser A)(p_2 : parser B)
  (q_1 : printer A)(q_2 : printer B),
  sequential p_1 \to sequential p_2 \to
  parser_parser p_1 q_1 \to parser_parser p_2 q_2 \to
  parser_parser (p_1 * p_2) (q_1 * q_2).

A parser and a printer that are related by the above relations
parser_parser and printer_parser, are indeed inverse to each other, and therefore unique when they exist. This unicity is
useful to convince oneself that a certain definition that might seem
odd at a first glance is in fact the right one. For instance, although
the definition of the combinator \( \text{Ret} \) as a parser is natural, one might
wonder if its definition as a printer is the right one. In fact this is
the unique one that satisfies the above relations parser Printer
and printer_parser.

4. Application to TLS packet processing

In this section, we apply our formalization of data-dependent
invertible syntax descriptions (explained in Sect. 3) to the formaliza-
tion of TLS packets as described in the RFC [4]. We show that
data-dependent constraints are conveniently expressed using the
combinators we proposed. Sect. 4.2 illustrates data-dependent con-
straints depending on the parsed value, as found in most commu-
nication protocols. More specific to TLS are data-dependent con-
straints w.r.t. the input list of bytes that are illustrated in Sect. 4.3.
First, we start with a technical overview of TLS and its RFC.

4.1 Overview of the TLS protocol

The TLS protocol provides privacy and data integrity to commu-
nication applications by adding a cryptographic layer on top of
TCP [17]. More precisely, the TLS protocol consists of four sub-
protocols (see Fig. 6). The Handshake protocol is for negotiating
a session. The Change Cipher Spec protocol is for signaling the
transition to the newly negotiated keys and parameters. The Alert
protocol is for dealing with exceptions such as deliberate interrup-
tion of the communication or errors. The Record protocol is used
by the above three protocols and the communication applications;
it serves as an intermediate with TCP and is in charge of fragmenta-
tion, compression, encryption and adding a MAC.

4.2 Data-dependent constraints on the parsed value

A grammar rule for parsing (resp. printing) the next bytes may
depend on previously parsed (resp. printed) values. Let us illustrate

Figure 5. The exa combinator

Figure 6. Transport Layer Security (TLS)
this with TLS session identifiers. They are defined in the RFC as follows (Sect. 7.4.1.2 of [4]):

```plaintext
opaque SessionID:0..32;
```

As seen in Sect. 4.1, this means that a session identifier is a list of bytes, whose length lies between 0 and 32, and that is preceded with a header containing the precise length in question.

Fig. 7 displays the grammar rule `SessionID_syntax` for parsing (resp. printing) TLS session identifiers, expressed with our formalization of invertible syntax descriptions. Since the length of a session identifier is variable, we resort to data-dependent parsing as provided by `repeat_dep` (Fig. 4). In order to rule out lists longer than 32 bytes, we use the guard combinator (Fig. 3) to parse the header. In Fig. 7, `SessionID` is the type for session identifiers in the abstract syntax. It is defined as being a list of bytes whose length is not constrained explicitly but we can prove the following: (1) if this list is the result from the parser `SessionID_syntax` then it will be of length at most 32, and (2) the printer `SessionID_syntax` will fail when applied to a list of length greater than 32.

As another example of data-dependent constraints w.r.t. the parsed value, Fig. 8 provides the grammar rule `Handshake_body_syntax` to parse (resp. print) Handshake packets (that we used as a motivating example in the Introduction of this paper). The type of the field body of a Handshake packet explicitly depends on the value of the field `msg_type`; there is also an implicit dependency between the length encoded in the field `h_length` and the function to parse (resp. print) the body. The purpose of the partial isomorphism `record_Handshake` is to transform the dependent pair returned by `Prod_dep` into a dependent record of type `Handshake`. All partial isomorphisms whose name is prefixed by `record_` serve similar purpose.

The next section explains `ClientHello_syntax` as a concrete example of a body of a Handshake packet.

### 4.3 Data-dependent constraints on the input bytes

The previous section (Sect. 4.2) gave grammar rules with data-dependent constraints where the constraint was expressed w.r.t. the parsed value. It turns out that the processing of network packets also requires parsing with data-dependent constraints where the constraint is expressed w.r.t. the number of bytes used for parsing (resp. printing) previous values. In the case of TLS, this is exemplified by a Handshake packet whose field body is of type `ClientHello`:

```plaintext
Record ClientHello : Type := {
    client_version : ProtocolVersion;
    random : Random;
    session_id : SessionID;
    cipher_suites : list CipherSuite;
    compression_methods : list CompressionMethod;
    extensions : option (list byte)
}.
```

The field of interest is the optional extensions field. At parsing, its presence or absence is decided after parsing up to the field `compression_methods`. If the value of the field `h_length` of the Handshake packet is greater than the number of byte processed so far, then extensions are present and their length is the difference between `h_length` and the number of processed bytes. In the RFC [4] for TLS it is specified as follows:

“*The presence of extensions can be detected by determining whether there are bytes following the compression methods at the end of the ClientHello. Note that this method of detecting optional data differs from the normal TLS method of having a variable-length field, but it is used for compatibility with TLS before extensions were defined.*”

Let us stress it again. The data-dependency here is different from data-dependence constraints w.r.t. a previously parsed (resp. printed) value: we are here concerned with the number of previously parsed (resp. printer) bytes. This is this difference that motivates the introduction of the combinators `Len` and `ens` (Sect. 3.2.2). We now show how to use them to parse (resp. print) `ClientHello` packets.

We now explain the grammar rule for `ClientHello` packets (Fig. 9) from the point of view of parsing. We omit the definitions of the simpler `ProtocolVersion_syntax` (line 5) and `Random_syntax` (line 6) grammar rules. `SessionID_syntax` (line 7) has been explained in Sect. 4.2. The grammar rules for `cipher_suites` and `compression_methods` (lines 8 and 9) are similar to the grammar rule `SessionID_syntax` but require additional tests to be performed...
Figure 9. ClientHello packets: Grammar rule to on the left, RFC specification on the right

Specifically, these variable-length vectors must not be empty (hence the guard test) and their length must be a multiple of the length of their elements (hence the divn_iso test). We now come to the extensions field and data-dependent constraint w.r.t. the input list of bytes. In the case of the parsing of a Handshake packet with a ClientHello body, the combinator len is used to determine the length of the body (line 4). By using Prod_dep, this length is passed to the next grammar rule to determine the presence or not of extensions (line 11). Finally, the combinator exa (line 3) is used to ensure that the length of the packet is the one specified in the field h_length (that comes from the outer packet).

The explanations we gave in the previous paragraph were from the point of view of parsing, where the presence/absence of extensions is decided w.r.t. the length passed from the outer packet (the field h_length of the Handshake packet, passed to ClientHello syntax). From the point of view of printing, the formalism of invertible syntax descriptions requires us to also pass this length, which is redundant since it can be computed from the AST. We defer to future work an improvement of invertible syntax to avoid this redundancy.

5. Practical Considerations

Automation Each time we define a new grammar rule by using combinators, we prove that the relations parser_parser, printer_parser and sequential hold (cf. Sect. 3.3). Those proofs are automatic thanks to the tactic auto of Coq that works with the dedicated libraries of hints that we have built for that purpose. Since those proofs are automatic, one might wonder whether they could not be replaced by one general theorem that would state that those properties hold by construction. Indeed, (1) they are true of basic combinators Tok and Ret, and (2) they are preserved by the other combinators. It is however not possible to state such a theorem because there is no syntax for combinators (they are shallow embedded in Coq).

The specification of a protocol such as TLS includes long lists of constants and their encoding. This might lead to oversights when inserting these lists in the formalization. In order to avoid such a problem, we propose to use a script that generates inductive types for these lists and the partial isomorphisms between values in these lists and their encoding in bytes. We have done that for TLS cipher suites and their encodings as two bytes.

Extraction of reference implementations Using the extraction mechanism of Coq, it is possible to obtain functional programs (in, say, O'Caml) from the grammar rules formalized using invertible syntax descriptions. Even though extracted programs can be compiled to native code, such reference implementations are unlikely to be efficient and used in real applications. Yet, they provide solid references to test real implementations. In the course of formalizing TLS packet processing, we have developed such a testing infrastructure and confirmed proper Handshakes with the most widely used implementation of TLS (namely, OpenSSL [22]).

Extraction is automatic but there are some caveats. Since the input channel is modeled in Coq as a list, we need to substitute access to the input bytes by reading out of a socket. The only combinator that accesses to the input list of bytes is Tok. Its instance as a parser accesses to the input bytes through the function read_byte that simply get the first byte of byte list. In O'Caml this function is extracted as a function that reads a byte on the input channel.
In order to locate syntax errors when a packet is not syntactically correct, we make use of the following function in the formalization:

Definition DEBUG {A : Type} (s : string) (v : A) : A := v.

Its parameter s is ignored in Coq; but in O’Caml, DEBUG is extracted as a function that prints the string s and returns the parameter v.

6. Work in Progress

Relations between parsing and printing In order to accommodate more grammars, it is possible to weaken the relations between parsing and printing introduced in Sect. 3.3 so that parsing followed by printing of a string can lead to an output different from the input. As already hinted at in Sect. 3.3, this situation occurs in the case of programming languages or textual protocols, where parsed blanks might be printed differently. This can also be caused by padding in the case of binary protocols. In the case of a data format including a compression scheme such as DNS, there might also be different encodings for the same data. Even in these situations, the following relation is still satisfied: parsing, printing, and then parsing again is the same as parsing only once:

Definition parser_printer_weak {A : Type} (p : parser A) (q : printer A) : Prop :=
\forall s_1 \ s_2 \ a, \ p (s_1 \ ++ \ s_2) = \text{Some} (a \ s_2) \rightarrow\ q a = \text{Some} s_1 \rightarrow \ p (s_1 \ ++ \ s_2) = \text{Some} (a \ s_2).

The stronger version parser_printer introduced in Sect. 3.3 however holds for a binary protocol like TLS because there are no blanks and padding is deterministic.

Similarly, the printer_parser relation introduced in Sect. 3.3 cannot be proved in general: for example, consider a parser for arithmetic expressions and take s to be the expression "1 + 2" and s2 to be the expression "x 3", where x has precedence over +. The printer_parser definition can however be weakened as follows:

Definition printer_parser_weak {A : Type} (p : parser A) (q : printer A) : Prop :=
\forall s \ a, \ q a = \text{Some} s \rightarrow \ p s = \text{Some} (a, \text{nil}).

The stronger version printer_parser introduced in Sect. 3.3 however holds for TLS because there is no notion of precedence in its grammar since it is a binary protocol.

Relations between combinators In the process of simplifying the formalization, we have been investigating relations between combinators. Such relations can be extensions to the class Syntax (Fig. 2), e.g., the relation between the composition of partial isomorphisms (noted g o f) and the Map_combinator:

Map_comp : \forall A B C (f : Iso A B) (g : Iso B C) p, Map (g o f) p = Map g (Map f p);

Such relations can also be lemmas external to the class Syntax that express rewriting rules between combinators, e.g., the fact that repeat (Fig. 5) can be expressed with Many under appropriate hypotheses:

Variable T : Type \rightarrow Type.
Hypothesis S : Syntax T.
Variable A : Type.
Hypothesis E : EqDec A eq.

Lemma exa_Many_repeat (p : T Z) (q : T A) (nb size : nat) :
1 \leq size \rightarrow q = exa (Z_of_nat size) q \rightarrow Many (nb \* size) q = repeat nb q.

We believe that there is much value in identifying and exploiting a set of such relations but, at the time of this writing, this is still work in progress.

7. Related work

In this paper, we extend invertible syntax descriptions [18] to deal with data-dependencies. The treatment of such context-sensitive restrictions is the subject of much related work: the meta-\( \lambda \) calculus [9], YAKKER [10, 11], etc. Among them, [11] provides a comprehensive presentation of parsing using data-dependent grammars, complemented by an efficient implementation (efficiency is achieved by allowing reuse of existing implementations for context-free grammars that have been optimized over the years) [10]. To compare, our work addresses two complementary issues: the printing of context-sensitive syntax and machine-checked proofs.

In this paper, we not only extend invertible syntax descriptions [18] to deal with data-dependencies but also formalize them in the Coq proof assistant. Similarly, a dependently-typed data description language is given semantics both as parser and pretty-printer in [14]. Also, [16] proposes a uniform approach to parsing and printing by providing generic parsers and printers for a data description language embedded in a dependently-typed language, namely Agda (see Sect. 3 of [16]). This approach is further investigated and extended in [2] by dealing with the details of bit-level processing for network packet processing. Our work goes one step further by establishing formally the relationship between parsing and printing.

There is also a large body of papers about bidirectional languages for which the relation between parsing and printing is an illustration of choice. For instance, in [5] and [13], languages are proposed that allow for unifying parsing and printing, but the authors do not go as far as to embed them in a proof-assistant. Lenses, and in particular string lenses, are bidirectional transformation (see for example [1]). However it should be pointed out that bidirectionality is not the same as invertibility.

8. Conclusion

In this paper, we have extended invertible syntax descriptions so that they can deal with data-dependent grammars and we have formalized them as a library of functions in the Coq proof assistant about which we also specify and prove their properties. Using the extraction mechanism of Coq, we confirmed that one can obtain certified implementations of parsers and printers; since the grammar is specified only once and is used both for parsing and printing, such reference implementations are free of inconsistencies that could occur with the usual approach where the parser and the printer are developed separately. We have demonstrated the usefulness of the resulting library by specifying network packet formats for the TLS protocol. This experiment showed in particular that data-dependent grammars are not only useful when the data-dependent constraint is expressed w.r.t. the parsed value, but also when it is expressed w.r.t. the input list of bytes.

References


