Encoding and Decoding Network Packets in Coq

David Nowak

JFLI, CNRS & The University of Tokyo

4th Asian-Pacific Summer School on Formal Methods
ECNU, Shanghai
July 20, 2012

The work presented here was done while the author was affiliated to AIST, and was supported by NICT. It appeared in the proceedings of PLPV 2012.

Introduction

- **Topic of this presentation:**
  formalization of parsing (decoding) and printing (encoding) of network packets.

  > An important part of the specification for an Internet protocol is about the syntax of network packets.

- **Approach:** a formalization method (in Coq) that:
  1. avoids parsing/printing inconsistencies, and
  2. deals with context-sensitive grammars.

- **Motivating example:** TLS (Transport Layer Security)

- **Why you should listen at this talk?**
  1. It is a simple application of advanced features of Coq: dependent types and type classes.
  2. It is a concrete application, not a toy example;
  3. yet requires only a basic background in computer science.

Outline

Parsing and printing

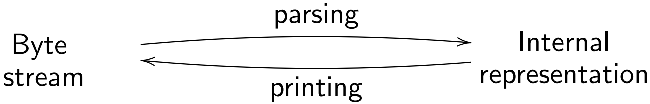
The need for data-dependent parsing/printing with TLS

Application to TLS

Summary

Outline

The problem with parsing and printing



- Parsing and printing are almost inverse to each other.

- However, they are usually implemented separately.
  - ⇒ Redundancy

    The grammar has to be specified twice (for parsing and printing).
  - ⇒ Potential inconsistencies

    How do I know that I have specified twice the same grammar?!

Parsing with Yacc

- Automatic generation of a parser from a grammar

$$\text{Grammar} \xRightarrow{\text{Yacc}} \text{Parser}$$

- Suitable for programming languages implementations
- Lacks support for context-sensitive grammars needed for:
  - data formats
  - networking protocols
  - configuration files
  - domain-specific languages
  - ...
- In fact, most parsers are done by hand in an ad-hoc way without any formal grammar.

Parsing with a monad

- Popular approach to parsing in functional programming
  - Parsers are functions:
    Input: a list of bytes
    Output: the parsed value and the remaining bytes
  - Grammar constructions are combinators (higher-order functions): sequencing, choice, repetition. . .
- They form an instance of a monad:
  - A monad is an algebraic structure that has proved useful to model imperative features in purely functional languages (such as the language of Coq).

    Being a monad guides the design of the combinators library.
- Allows both for:
  - a clean presentation of the grammar, and
  - the flexibility of a programming language.

Printing: no ready-to-use solution

- No counterpart of Yacc in imperative programming

- There are combinators library in functional programming.
  - They are ad hoc (not a monad).
  - They give printers that are independent of the parsers (redundancy, possible inconsistencies).

Invertible Syntax Descriptions

- Introduced by Rendel and Ostermann in 2010
- Allow to make a parser and printer at the same time:
  - $\Rightarrow$ No redundancy, no inconsistencies
- A collection of combinators to describe grammars:

```
Class Syntax (T : Type → Type) := {
 Tok : T byte ;
 Ret : ∀ {A : Type}{_ : EqDec A eq}, A → T A ;
 Fail : ∀ (A : Type), T A ;
 Map : ∀ {A B : Type}, Iso A B → T A → T B ;
 Prod : ∀ {A B : Type}, T A → T B → T (A ∗ B) ;
 Many : ∀ {A : Type}, nat → T A → T (list A) ; ... }
```

- The above class is instantiated twice (i.e., the combinators are overloaded):

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

- In this formalization, each combinator has two meanings (parser/printer) that are proved “inverse” to each other.

Parser and printer as inverses to each other

(i) If one parses a list of bytes s_1 into a value a, then one gets back the list s_1 when printing a:

```
Definition parser_printer {A : Type}
 (p : parser A) (q : printer A) : Prop :=
 ∀ s₁ s₂ a, p (s₁++s₂) = Some (a, s₂) → q a = Some s₁.
```

incompatible with optional blanks: $s_1 =$ " x   + 4"

(ii) If a value a is printed into the list of bytes s_1 that is then parsed in a larger context where s_1 is followed by a further list s_2, 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 :=
 ∀ s₁ s₂ a, q a = Some s₁ → p (s₁++s₂) = Some (a, s₂).
```

Incompatible with priorities: $s_1 =$ "1+2" and s_2="*3"

- We prove formally that (i) and (ii) hold for each of the combinators.
- As expected of an inverse, it is unique when it exists.

Tok: parsing/printing a single token

- As a parser, `Tok` returns the head of the input list of bytes, failing when the latter is empty:

```
Instance Syntax_parser : Syntax parser := {
  Tok := λ s ⇒
    match s with nil ⇒ None | b :: s' ⇒ Some (b, s') end ;
... }
```

- As a printer, `Tok` inputs one byte that it returns as a singleton list:

```
Instance Syntax_printer : Syntax printer := {
  Tok := λ b ⇒ Some [b];
... }
```

Ret: trivial parser/printer

- As a parser, `Ret a` does not consume any byte but always returns successfully the value `a`:

```
Instance Syntax_parser : Syntax parser := {
...
  Ret a := λ s ⇒ Some (a, s);
... }
```

- As a printer, `Ret a` only accepts the value `a` as input and prints the empty string, while failing on any other value:

```
Instance Syntax_printer : Syntax printer := {
...
  Ret a := λ x ⇒ if equiv_dec a x then Some nil else None;
... }
```

Fail: failure

- As a parser, `Fail` always fail:

```
Instance Syntax_parser : Syntax parser := {
...
  Fail := λ s ⇒ None;
... }
```

- As a printer, `Fail` always fail:

```
Instance Syntax_printer : Syntax printer := {
...
  Fail := λ x ⇒ None;
... }
```

Prod: parsing/printing sequences

`Prod p q` applies the parser (resp. printer) `p`, and then `q`.

- Instance Syntax_parser : Syntax parser := { ...
```
  Prod p q := λ s ⇒ match p s with
          | None ⇒ None
          | Some (a, s') ⇒ match q s' with
               | None ⇒ None
               | Some (b, s'') ⇒ Some ((a,b), s'')
               end
          end; ... }
```

- Instance Syntax_printer : Syntax printer := { ...
```
  Prod p q := λ ab ⇒ match p (fst ab) with
          | None ⇒ None
          | Some l1 ⇒ match q (snd ab) with
               | None ⇒ None
               | Some l2 ⇒ Some (l1++l2)
               end
          end; ... }
```

printer_parser and *parser_printer* for Prod

- The relation *printer_parser* holds for `Prod`.

```
Notation "p ∗ q" := (Prod p q).

Lemma Prod_printer_parser : ∀ A B
(p1:parser A)(p2:parser B)(q1:printer A)(q2:printer B),
printer_parser p1 q1 → printer_parser p2 q2 →
printer_parser (p1 * p2) (q1 * q2).
```

- But we need sequentiality to prove that *parser_printer* holds.

```
Definition sequential {A : Type} (p : parser A) : Prop :=
∀ s s2 a, p s = Some (a, s2) → ∃s1, s = s1++s2.

Lemma Prod_parser_printer : ∀ A B
(p1:parser A)(p2:parser B)(q1:printer A)(q2:printer B),
sequential p1 → sequential p2 →
parser_printer p1 q1 → parser_printer p2 q2 →
parser_printer (p1 * p2) (q1 * q2).
```

Many: repeating for a certain length (1/2)

As a parser, `Many n p` consumes exactly `n` bytes to parse a list of elements with the parser `p`:

```
Fixpoint Many_rec_parser (A:Type)(n:nat)(p: parser A)(s: list byte)
  {measure (λ x ⇒ x) n} : option ( list A ∗ list byte) :=
  match n with
  | O ⇒ Some (nil, s)
  | S _ ⇒ match p s with
    | None ⇒ None
    | Some (a, s') ⇒
      if andb (leb 1%nat (length s − length s'))
              (leb (length s − length s') n) then
      match Many_rec A (n − (length s − length s'))%nat p s' with
      | None ⇒ None
      | Some (l, s'') ⇒ Some (a::l, s'')
      end else None
    end
  end.

Instance Syntax_parser : Syntax parser := { ...
  Many := Many_rec_parser; ...}
```

Many: repeating for a certain length (2/2)

As a printer, `Many n p` prints a list of elements with the printer `p` to form a list of exactly `n` bytes:

```
Fixpoint Many_rec_printer (A:Type)(n:nat)(p: printer A)(al: list A)
  {struct al} : option (list byte) :=
  match n, al with
  | O, nil ⇒ Some nil
  | S _, a :: al' ⇒ match p a with
    | None ⇒ None
    | Some l1 ⇒ if andb (leb 1 (length l1)) (leb (length l1) n) then
      match Many_rec_printer A (n − length l1)%nat p al' with
      | None ⇒ None
      | Some l2 ⇒ Some (l1++l2)
      end else None
    end
  | _, _ ⇒ None
  end.

Instance Syntax_printer : Syntax printer := { ...
 Many := Many_rec_printer; ...}
```

printer_parser and *parser_printer* for Many

- The relation *printer_parser* holds for `Many`.

```
Lemma Many_printer_parser :
  ∀ A n (p : parser A)(q : printer A),
  printer_parser p q →
  printer_parser (Many n p) (Many n q).
```

- We need sequentiality to prove that *parser_printer* holds.

```
Lemma Many_parser_printer :
  ∀ A n (p : parser A)(q : printer A),
  sequential p → parser_printer p q →
  parser_printer (Many n p) (Many n q).
```

Partial isomorphisms

- Partial isomorphisms are defined by:

```
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
}.
```

- **Example:**
  adding/removing an element to/from the beginning of a list

```
Program Definition cons_iso (A:Type) : Iso (A∗list A) ( list A) := {|
  apply := λ (a, l) ⇒ Some (cons a l);
  unapply := λ l ⇒ match l with nil ⇒ None | a::l' ⇒ Some (a,l') end
|}.
```

- Obligations proofs (for *apply_unapply* and *unapply_apply*) are automatically generated by Coq, and either proved automatically or interactively.

Map: for applying functions

- If `p : parser A` and `f : A → B`,
  `Map f p` (of type `parser B`) parses with `p` and then applies `f`.

```
Instance Syntax_parser : Syntax parser := { ...
  Map f p := λ s ⇒ match p s with
    | None ⇒ None
    | Some (a, s') ⇒ match apply f a with
        | None ⇒ None
        | Some b ⇒ Some (b, s')
      end
  end; ... }
```

- If `q : printer A` is the counterpart of `p`,
  `Map f q` (of type `printer B`) ...WHAT DO YOU THINK?...
  applies f^{-1} and prints with `q`.

```
Instance Syntax_printer : Syntax printer := { ...
  Map f p := λ b ⇒ match unapply f b with
    | None ⇒ None
    | Some a ⇒ p a
  end; ... }
```

Using Map for repeating and conditional parsing/printing

```
Variable T : Type→ Type.
Hypothesis S : Syntax T.
Variable A : Type.
Hypothesis E : EqDec A eq.

(* Combinator to repeat a grammar rule n times *)
Fixpoint repeat (n : nat) (p : T A) : T ( list A) :=
match n with O ⇒
| Ret nil
| S n' ⇒ Map (cons_iso _) (p * repeat n' p)
end.

(* Combinator to add a condition to a grammar rule *)
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 |}.

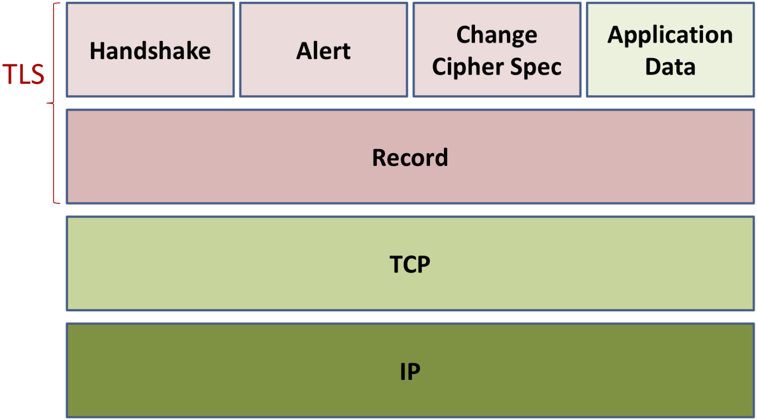
Definition guard (cond : A → bool) (p : T A) : T A :=
 Map (cond_iso cond) p.
```

- The above combinators have two possible meanings: parser or printer, depending on T.

Outline

Transport Layer Security (TLS)

A cryptographic layer on top of existing communication protocols



Need for data-dependent constraints on the parsed value

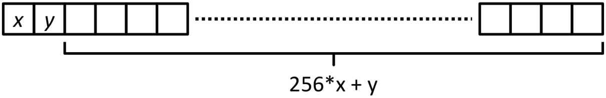
Examples:

- RFC 5246 (TLS) defines Handshake packets as follows:

```
struct {
  HandshakeType msg_type;    /* handshake type */
  uint24 length;             /* bytes in message */
  select (HandshakeType) {
      case hello_request:       HelloRequest;
      case client_hello:        ClientHello;
      ...
 } body;
} Handshake;
```

  The above is actually a **dependent record**: the type of the last field body depends on the value of the first field msg_type.

- In a variable-length field, the number of bytes depends on the value of the first bytes:



Need for data-dependent constraints on the input bytes

The grammar rule for parsing the next bytes may depend on the number of bytes used for parsing the previous value.

```
struct {
  ProtocolVersion client_version;
  Random random;
  SessionID session_id;
  CipherSuite cipher_suites<2..2^16-2>;
  CompressionMethod compression_methods<1..2^8-1>;
  select (extensions_present) {
      case false:
          struct {};
      case true:
          Extension extensions<0..2^16-1>;
  };
} ClientHello;
```

"*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.*" (Extract from RFC 5246)

Prod_dep: data-dependent parsing/printing

`Prod_dep p q` applies the parser (resp. printer) `p`, and then `q a` where `a` is the parsed/printed value by `p`.

- ```
  Class Syntax (T : Type → Type) := { ...
   Prod_dep : ∀ {A:Type}{B:A→ Type},
    T A → (∀ a, T (B a)) → T {a:A & B a}; ...}
  ```
- ```
  Instance Syntax_parser : Syntax parser := { ...
   Prod_dep p q := λ s ⇒ match p s with
                    | None = None
                    | Some (a, s') ⇒ match q a s' with
                      | None ⇒ None
                      | Some (b, s'') ⇒ Some (existT (λ a ⇒ B a) a b, s'')
                      end
                  end; ... }
  ```
- ```
  Instance Syntax_printer : Syntax printer := { ...
   Prod_dep p q := λ ab ⇒ match p (projT1 ab) with
                    | None ⇒ None
                    | Some l1 ⇒ match q (projT1 ab) (projT2 ab) with
                      | None ⇒ None
                      | Some l2 ⇒ Some (l1++l2)
                      end
                   end; ... }
  ```
  ```

Example of use of Prod_dep

```
Variable T : Type→ Type.
Hypothesis S : Syntax T.
Variable A : Type.
Hypothesis E : EqDec A eq.

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 l ⇒ Some (ℤ_of_nat (length l), l) |}.

Program Definition undep_iso : Iso {_:A & B} (A∗B) := {|
apply := λ x ⇒ Some (projT1 x, projT2 x);
unapply := λ len ⇒ Some (existT _ (fst len) (snd len)) |}.

(∗ A combinator to specify grammar rule for variable−length list ∗)
Program Definition repeat_dep
(p1 : T ℤ) (p2 : T A) : T ( list A) :=
 Map ((chop_len_iso A) o (undep_iso _ _))
  (Prod_dep p1 (λ n ⇒ repeat (ℤabs_nat n) p2)).
```

Len: number of parsed/printed bytes

```
Class Syntax (T : Type → Type) := { ...
Len : ∀ {A:Type}, T A → T (A*ℤ); ...}
```

- As a parser, `Len p` extends the parser `p` such that it does not only return the parsed value, but also the number of input bytes consumed to parse this value.

```
Instance Syntax_parser : Syntax parser := { ...
 Len p := λ s ⇒ match p s with
        | None ⇒ None
        | Some (a, s') ⇒ Some ((a, ℤ_of_nat (List.length s − List.length s')), s')
        end; ... }
```

- As a printer, when applied to a pair `(a, len)`, `Len p` prints the value `a` if it can be printed as `len` bytes, and fails otherwise.

```
Instance Syntax_printer : Syntax printer := { ...
 Len p := λ a_len ⇒ match p (fst a_len) with
        | None ⇒ None
        | Some l ⇒
           if ℤeq_bool (ℤ_of_nat (List.length l)) (snd a_len)
           then Some l
           else None
        end; ... }
```

The derived combinator exa

We define `exa` such that:
`exa n p` forces the parser/printer `p` to consume/print exactly `n` bytes.

- Program Definition proj_left_iso (b : B) :
  Iso (A $*$ B) A := {|
   apply := λ ab $\Rightarrow$
    if (snd ab) == b then Some (fst ab) else None;
   unapply := λ a $\Rightarrow$ Some (a, b)
  |}.
- Program Definition exa (n : $\mathbb{Z}$)(p : T A) : T A :=
  Map (proj_left_iso n) (Len p).
- Lemma exa_Many_repeat
  (p : T $\mathbb{Z}$) (q : T A) (nb size : nat) :
   1 $\leq$ size $\rightarrow$ q = exa ($\mathbb{Z}$_of_nat size) q $\rightarrow$
   Many (nb $*$ size) q = repeat nb q.

Outline

Session identifier

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.

- In RFC:

```
opaque SessionID<0..32>;
```

- In Coq:

```
Variable T : Type → Type.
Hypothesis S : Syntax T.

(* Internal representation of a session identifier *)
Definition SessionID : Type := list byte.

(* Syntax for a session identifier *)
Definition SessionID_syntax : T SessionID :=
 repeat_dep (guard (λ z ⇒ ℤle_bool z 32) Tok) Tok.
```

ClientHello sub-packet: Internal representation

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

ClientHello sub-packet: Syntax (but not too closely :-)

```
Variable T : Type → Type.
Hypothesis S : Syntax T.
Definition ClientHello_syntax (len : ℤ) : T ClientHello :=
 exa len (
  Map (record_ClientHello len) (Prod_dep (Len (
   ProtocolVersion_syntax *
   Random_syntax *
   SessionID_syntax *
   CipherSuites_syntax *
   CompressionMethods_syntax))
   (λ r ⇒
    if snd r == len then (* case false *)
     Ret None
    else if ℤlt_bool (snd r) len then (* error case *)
     DEBUG (" ClientHello_syntax " ++ string_from_ℤ
      (snd r) ++ " " ++ string_from_ℤ len) (@Fail _ _ _)
    else (* case true *)
     Map (Some_iso _ o chop_len_iso _ o undep_iso _ _)
      (Prod_dep int16_syntax
       (λ r ⇒ Many (ℤabs_nat r) Extension_syntax))))).
```

```
struct {

  ProtocolVersion client_version;
  Random random;
  SessionID session_id;
  CipherSuite cipher_suites<2..2^16-2>;
  CompressionMethod compression_methods<1..2^8-1>;

  select (extensions_present) {
      case false:
          struct {};
      case true:
          Extension extensions<0..2^16-1>;
  };


} ClientHello;
```

Handshake packet: Internal representation

```
Variable T : Type → Type.
Hypothesis S : Syntax T.

Inductive HandshakeType : Type :=
| hello_request | client_hello | server_hello
| certificate | server_hello_done.

Record Handshake : Type := {
  msg_type : HandshakeType;
  h_length : ℤ;
  body : HandshakeType_type msg_type }.
```

Handshake packet: Syntax of the body field

```
Definition HandshakeType_type
 (ht : HandshakeType) : Type :=
  match ht with
  | hello_request ⇒ HelloRequest
  | client_hello ⇒ ClientHello
  | server_hello ⇒ ServerHello
  | certificate ⇒ Certificate
  | server_hello_done ⇒ ServerHelloDone
  end.

Program Definition Handshake_body_syntax
 (ht : HandshakeType) (len : ℤ) :
 T (HandshakeType_type ht) :=
  match ht with
  | hello_request ⇒ HelloRequest_syntax
  | client_hello ⇒ ClientHello_syntax len
  | server_hello ⇒ ServerHello_syntax len
  | certificate ⇒ Certificate_syntax
  | server_hello_done ⇒ ServerHelloDone_syntax
  end.
```

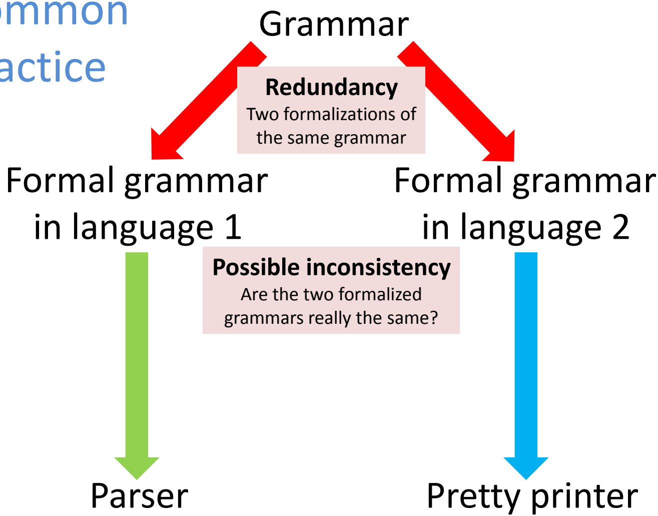
Handshake syntax: Syntax

```
Program Definition record_Handshake :
  Iso
    {ht:HandshakeType & {len:ℤ & HandshakeType_type ht}}
    Handshake := {|
  apply := λ r ⇒ Some {|
      msg_type := projT1 r;
      h_length := projT1 (projT2 r);
      body := projT2 (projT2 r) |};
  unapply := λ h ⇒
    Some (existT _ (msg_type h) (existT _ (h_length h) (body h))) |}.

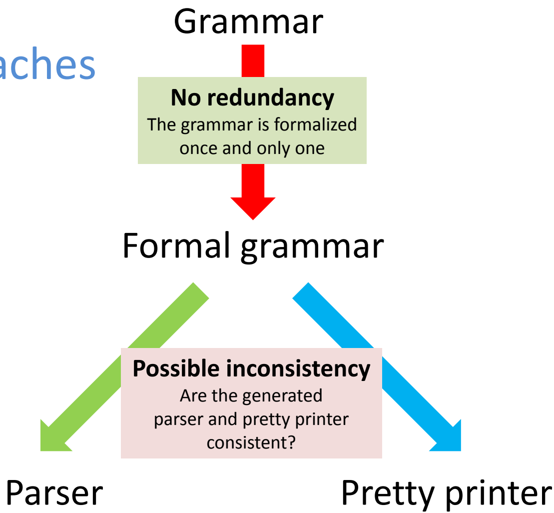
Program Definition Handshake_syntax : T Handshake :=
 Map record_Handshake
  (Prod_dep HandshakeType_syntax
   (λ ht ⇒ Prod_dep int24_syntax
    (λ len ⇒ Handshake_body_syntax ht len))).
```

Outline

Common Practice



Recent Approaches



Our Solution

Grammar

No redundancy
The grammar is formalized once and only one

Formal grammar

No inconsistency
The parser and pretty printer are guaranteed to be inverse to each other

Suitable for the formalization of binary protocols such as TLS but too strong for textual protocols

Parser

Pretty printer

Conclusions and future work

Summary

- New combinators for data-dependent parsing/printing

  dependency on the parsed value and on the input bytes
- Formalization in Coq:

  Allows for the extraction of a reference implementation.
- Formal relations between parsing and printing (as inverses)

  Proofs are automated.
- Application to TLS

What's next?

- Weaker relations to relate parsing and printing

  Useful to deal with textual protocols
- Equations between combinators

  Example: the relation between the composition of partial isomorphisms (noted g o f) and the Map combinator:

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

Reference

Formal network packet processing with minimal fuss: Invertible syntax descriptions at work.
Reynald Affeldt, David Nowak, and Yutaka Oiwa.
In *Proceedings of the 6th ACM Workshop Programming Languages meets Program Verification, PLPV 2012, Philadelphia, PA, USA, January 24, 2012*, pages 27-36. ACM.

`http://jfli.nii.ac.jp/medias/members/nowak/nowak-plpv2012.pdf`

Conclusion: Learning Coq (1/2)

- It is important to look again at the examples and exercises, as well as the Coq documentation.
- Pierre and Yves propose 200 (solved) exercises, at `www.labri.fr/perso/casteran/CoqArt/contents.html`
- Suscribe to the `coq-club` mailing list!
  - Don't hesitate to ask questions !
  - Look at the questions by other people, and at the answers.
  - Be the first to answer! It's easy: with time difference, you can answer while people in Europe are still sleeping.

Conclusion: Learning Coq (2/2)

- Look at the **user contributions** page in `coq.inria.fr`. You will find a lot of examples and tools on many domains : math, computer science, games, etc.
- Submitting your Coq development as a contribution provides visibility to your work and ensures that it will be made compatible with the forthcoming versions of Coq. For the Coq developers, it helps to evaluate the robustness and efficiency of the evolutions of Coq.
- Don't forget that using Coq is like a game: You want to be able to type `Qed` before 6 p.m., and the system wants your proof to be complete and correct.
- Quite often, the system helps you. It's a proof assistant.

Appendix: Syntax for a ClientHello sub-packet

```
Variable T : Type → Type.
Hypothesis S : Syntax T.
Definition ClientHello_syntax (len : ℤ) : T ClientHello :=
 exa len (
  Map (record_ClientHello len) (Prod_dep (Len (
   ProtocolVersion_syntax *
   Random_syntax *
   SessionID_syntax *
   CipherSuites_syntax *
   CompressionMethods_syntax))
   (λ r ⇒
    if snd r == len then (* case false *)
     Ret None
    else if ℤlt_bool (snd r) len then (* error case *)
     DEBUG ("ClientHello_syntax " ++ string_from_ℤ
      (snd r) ++ " " ++ string_from_ℤ len) (@Fail _ _ _)
    else (* case true *)
     Map (Some_iso _ o chop_len_iso _ o undep_iso _ _)
      (Prod_dep int16_syntax
       (λ r ⇒ Many (ℤabs_nat r) Extension_syntax))))).
```