パーサを用いた TLS 1.3の仕様書の検証

IIJ 技術研究所
山本和彦
@kazu_yamamoto

おことわり

今日はTLSとかRFCなどの
ネットワークの話ではありません

ネットワークを題材にした
プログラミングの話です

2008年8月にRFC 5246として
公開されたTLS 1.2は老朽化しました

4年かけて次世代TLSが議論され
2018年8月にTLS 1.3が
RFC 8446として公開されました

RFC 8446

The Transport Layer Security (TLS)
Protocol Version 1.3

...

Contributors

...
Kazu Yamamoto
Internet Initiative Japan Inc.
kazu@iij.ad.jp

「TLS 1.3の標準化と実装」

https://eng-blog.iij.ad.jp/archives/2056

パーサーを書けば、書式の間違いをたくさん発見できます。

パースした結果は、さらにHaskellのコードに変換しました。
Haskellはよく設計された言語であり、コンパイラを通すことで、
すべての場合が網羅されているかを検査できます。
この一手間を加えることで、重複や漏れを発見できます。

TLS 1.3の標準化で何をやったか

実装	相互接続生の検証
仕様の改善	曖昧な部分の改善 誤りの修正
書式の検証	文法の検証 パケットフォーマットの検証

この発表の趣旨

双対を理解して
プログラミングをうまくなる

双対？！

この発表での「双対」とは
「双子」あるいは「相棒」
ぐらいの意味です

TLS 1.3 のパケットフォーマットの例

- Cライクな文法が利用されている

```
uint16 ProtocolVersion;
opaque Random[32];
uint8  CipherSuite[2];

struct {
    ProtocolVersion legacy_version = 0x0303;
    Random random;
    opaque legacy_session_id<0..32>;
    CipherSuite cipher_suites<2..2^16-2>;
    opaque legacy_compression_methods<1..2^8-1>;
    Extension extensions<8..2^16-1>;
} ClientHello;
```

TLS 1.3 の文法定義

- 型の別名

```
T T';

uint8 uint16[2];
/* def uinit16 = uint8[2] なら分かりやすかったのに！ */
```

- 列挙型

```
enum { e1(v1), e2(v2), ... , en(vn) [[, (n)]] } Te;

enum { request(3), response(5), (255) } Type;
```

- 構造体

```
struct { T1 f1; T2 f2; ... Tn fn; } T;

struct {
    Type   type;
    uint16 length;
} Header;
```

RFCを書く人達は、このような
文法定義やパケットフォーマットを
手書きする。

機械的な検証はしない。

パーサを書けば
たくさんエラーを見つけられて
標準化に貢献できる！

パーサを実装する方法

正規表現 + ホスト言語	正規表現では入れ子構造を扱えない 正規表現は部品化が難しい
パーサジェネレータ	パーサジェネレータは自分の言語ではない
パーサコンビネータ	自分の言語のEDSL 部品化が容易 入れ子構造を扱える

ステップ1

- 文法定義に従ってパーサを書く

```
enum { request(3), response(5), (255) } Type;
```

- Haskell のパーサコンビネータ Parsec での例
  - 小さなパーサを連接して大きなパーサを作る

```
enum :: Parser DEF
enum = do
    reserved "enum"
    (lst,mx) <- braces $ do
        el <- endBy1 enumItem comma
        n <- parens natural
        return (el,n)
    name <- identifier
    semi
    return $ ENUM name mx lst
```

1つ目の双対

連接と選択

連接	A; B; C Aの次はB、Bの次はC
選択	A | B | C A または B または C

ステップ1 (続き)

- TLS 1.3 の文法
  - 型の別名
  - 列挙型 (`enum`)
  - 構造体 (`struct`)

- Parsecで選択コンビネータ `<|>` を使う例

```
defs :: Parser [DEF]
defs = many (enum <|> struct <|> alias)
```

パーサを実装するときに見つけた間違いたち

- ".." の意味が未定義

```
enum {
    obsolete_RESERVED (1..22),
    ...
    (0xFFFF)
} NamedGroup;
```

- 入れ子の struct は未定義

```
struct {
    select (Handshake.msg_type) {
      ...
      case server_hello: struct {};
    };
} EarlyDataIndication;
```

ステップ2

- パーサでパケットフォーマットをパースする
  - 間違っているパケットフォーマットを発見する

パーサで見つけた間違いたち

- カンマやセミコロンがない

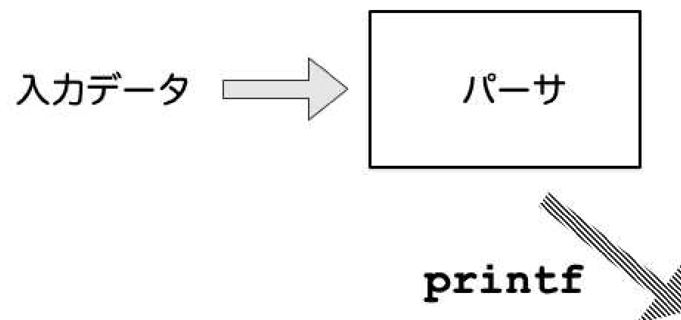
```
struct {
    select(certificate_type) {
      ...
      case X509: opaque cert_data<1..2^24-1>;
    }
    Extension extensions<0..2^16-1>;
} CertificateEntry;
```

- case の中身が複数

```
struct {
    select (Handshake.msg_type) {
      case client_hello:
          PskIdentity identities<6..2^16-1>;
          PskBinderEntry binders<33..2^16-1>;
      ...
    };
} PreSharedKeyExtension;
```

命令型言語のパーサあるある

- パーサが主役
- パースしながら何かを出力する
  - よく言えばストリーム処理



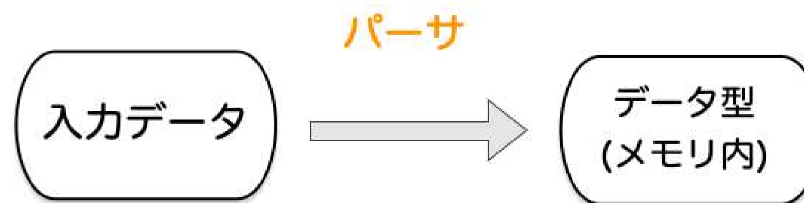
関数型言語ではデータ型が主役

- TLS 1.3の文法定義を表す Haskell の型

```
data DEF = ENUM   DEFNAME Int [ENUMITEM]
         | STRUCT DEFNAME [MEMBER]
         | ALIAS  TYPE

data TYPE = TYPE NAME TYPENAME AMOUNT

type NAME = String

type TYPENAME = String

data AMOUNT = SCALAR
            | VECTOR  Int     -- Int = byte count
            | VVECTOR Int Int -- min .. max
```

関数型言語のパーサ

- パーサは単なる変換関数
  - あくまでデータが主役



2つ目の双対

プリティプリンタ

- プリティプリンタも変換関数

ステップ３

- プリティプリンタを実装する
  - HaskellのデータDEF → Haskellのプログラム

```
pp (ALIAS (TYPE new old _)) = do
    let old’ = cap old
    putStr "newtype "
    putStr new
    putStr " = "
    putStr new
    putStr " "
    putStr old’
    putStr "\n"
```

- 変換前のTLS 1.3の定義

```
uint16 ProtocolVersion;
```

- 変換後のHaskellのコード

```
newtype ProtocolVersion = ProtocolVersion Uint16
```

ステップ4

- 出力されたHaskellのコードをHaskellコンパイラに通す
  - 定義が重複している
  - 定義されているのに使われていない
  - 定義されてないのに使われている

コンパイラが見つけた間違い

- 定義されているのに使われていない
- 定義されていないのに使われている

```
opaque ASN1Cert<1..2^24-1>;

struct {
    ASN1CertData cert_data;
    Extension extensions<0..2^16-1>;
} CertificateEntry;
```

3つ目の双対

直積と直和

直積

複数のメンバーを同時に持つ型

C の `struct`

オブジェクト

直和

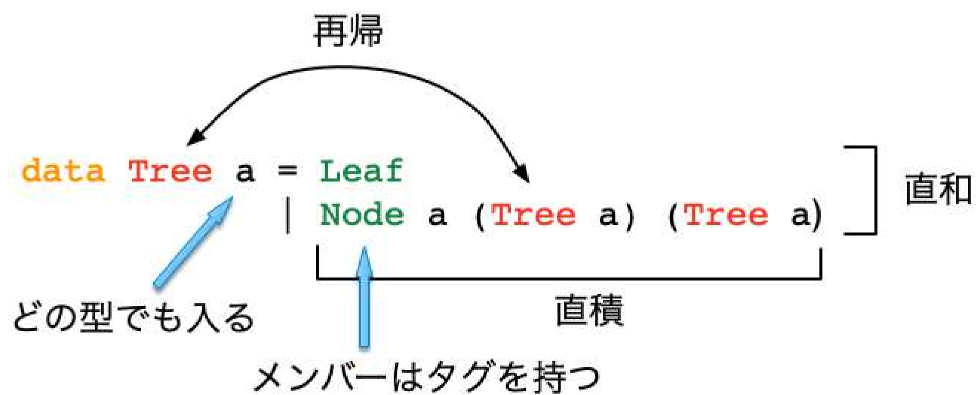
どれか1つになる型

C の `union`

Java の `enum`

関数型言語のデータ構造

- 直積の直和 (再帰あり)
- 木構造の例



直和とパターンマッチ

- 木構造 再掲

```
data Tree a = Leaf
            | Node a (Tree a) (Tree a)
```

- パターンマッチによる網羅検査
  - 漏れや重複はコンパイラが警告する

```
case tree of
  Leaf       -> ...
  Note x l r -> ...
```

網羅検査

- TLS 1.3

```
struct {
    ...
    select (Handshake.msg_type) {
      case client_hello:       ClientHello;
      case server_hello:       ServerHello;
      case end_of_early_data: EndOfEarlyData;
    ...
    };
} Handshake;
```

- Haskell で網羅を確認するコード

```
_a3424017607323890874 :: ()
_a3424017607323890874 = case undefined of
    CLIENT_HELLO      -> seq ClientHello{}    ()
    SERVER_HELLO      -> seq ServerHello{}    ()
    END_OF_EARLY_DATA -> seq EndOfEarlyData{} ()
    ...
```

- 漏れがないことを確認できた

まとめ

- TLS 1.3 の標準化に貢献した
  - パーサを実装し、文法やパケットフォーマットの誤りを発見
  - プリティプリンタを実装し、不整合を発見
- 意識した双対
  - 連接と選択
  - 直積型と直和型
  - パーサとプリティプリンタ

（いらすとや)