逆方向実行可能言語Conservationによるエンコーダとデコーダの同時実装

田中 哲

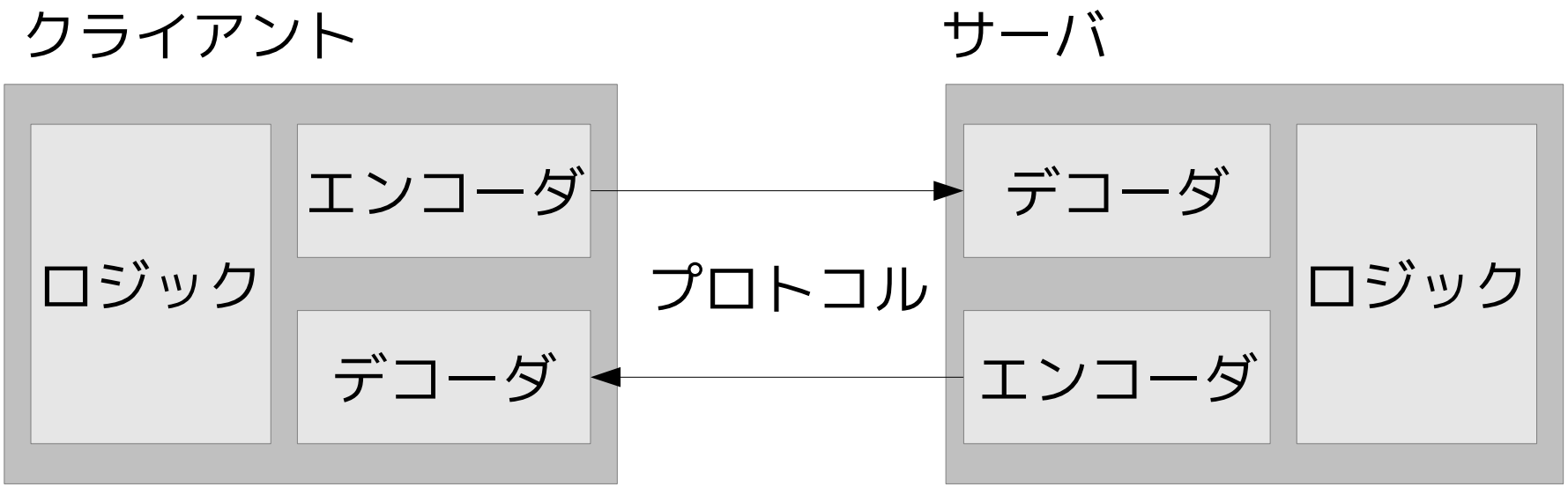
2014-01-14

産業技術総合研究所セキュアシステム研究部門

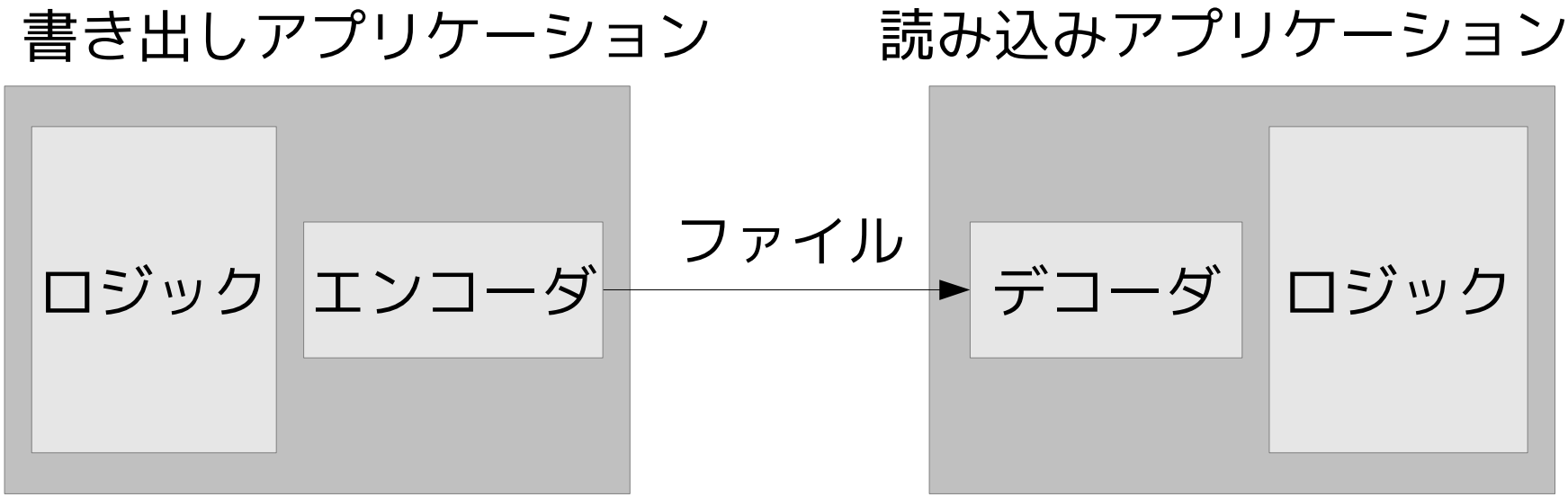
概要

- ひとつの記述からエンコーダとデコーダを生成
  記述が少なければバグも少ない
- 逆方向実行可能言語Conservationの提案
- 例として整数と文字列の変換を記述
- 例としてX.509証明書のエンコーダとデコーダを記述
- 応用として X.509 証明書のファジング
  OpenSSL の X.509 証明書の実装をテスト

通信プロトコルの エンコーダとデコーダ



ファイルフォーマットの エンコーダとデコーダ



エンコーダとデコーダは逆関数

- Encode: Tree → String
- Decode: String → Tree
- Decode(Encode(T)) = T
- 加えて Encode(Decode(S)) = S を仮定
  （一般には、この仮定は自然でないけれど）

原理的には、ひとつの記述で両方を実現できるはず

エンコーダとデコーダのための逆方向実行可能プログラミング言語

- 情報を失わず、副作用無く順方向実行できる
- 決定的に逆方向実行できる（Prologとの違い）
- 再帰下降構文解析のために再帰とローカル変数を使える（Janus との違い）
- String と Tree のようなデータ構造を扱える（Janus との違い）

新しい言語 Conservation の提案

Conservationによる自然数と文字列の変換

```
define n -> nat2str() -> s
  cond
  when { n -> !bind(0); }
    {}
    { bind([]) -> s; }
  when { guard(0 < n); }
    {
      n -> divmod(10) -> q, r;
      q -> nat2str() -> t;
    }
    { r, t -> cons() -> s; }
  end
end
```

逆方向実行による制約

通常のプログラミング言語の要素はたいてい逆方向実行可能ではない

例:

- 変数
- 条件分岐
- 関数呼び出し
- プリミティブ関数

変数

- 変数に代入すると元の値が消える（情報を失う）
  → 代入しなければよい (副作用がなくても困りませんよね)
- 関数から出たときにローカル変数が消える（情報を失う）
  → 出るときに値をどこかに記録すればよい
- 逆方向実行可能なローカル変数 Y のライフサイクル

  Y = X　(定義)
  … Y …　(使用)
  … Y …　(使用)
  Z = Y　(消費)　　変数Yを消費し、変数Zに記録する
- 記録の方法: 他の変数への代入、関数からの返値など
- 定義と消費は逆方向実行では反転する

注: Conservation の記法ではない

条件分岐

- 逆方向に実行すると条件分岐でどちらを実行したらいいかわからない
  → 逆方向に実行したときのための条件式を記述させる
- 逆方向実行可能なif文:
  if pred1
  then statement
  else statement
  endif pred2
- 順方向: pred1を評価してthen節かelse節のどちらかを実行
  pred2はどちらを実行したかの表明
- 逆方向: pred2を評価してthen節かelse節のどちらかを実行
  pred1はどちらを実行したかの表明
- 一般に、制御が合流するときには表明が必須

注: Conservation の記法ではない

関数呼び出し

- 関数の出口は一ヶ所でなければならない
  出口が二ヶ所以上あると逆方向でどこから実行したらいいかわからない
- 関数はローカル変数をすべて呼び出し元に返さなければならない
  逆方向でローカル変数の値がわからない
- 末尾再帰はできない
  - ある方向で末尾ならば、反対方向では末尾でない
  - 末尾再帰はgotoで、gotoは制御の合流
    (再帰にはの終了条件の分岐があり、再帰から帰ってきた後に表明の検査があって末尾位置での呼び出しにならない)

プリミティブ関数

- プリミティブ関数も逆方向実行可能でなければならない
- cons は逆方向実行可能
  順方向: X = cons(Y, Z)
  逆方向: Y = car(X); Z = cdr(X)
- プリミティブ関数は失敗しうる

  Xが空リストのとき、consを逆方向実行すると失敗する

注: Conservation の記法ではない

プリミティブ関数 (2)

- 加算は逆方向実行可能でない
  X = Y + Z
- 使用する引数と消費する引数を指定すれば、加算を逆方向に実行できる
  順方向: X = Y + Z　　(Y:使用, Z:消費)
  逆方向: Z = X - Y

注: Conservation の記法ではない

プリミティブ関数 (3)

- divmod は逆方向実行可能
  被除数 N を消費、除数 M を使用とする
  順方向: Q, R = divmod(N, M)
  逆方向: N = Q*M+R
- divmod はビットの取り出しに利用可能
  hibits, lsb = divmod(N, 2)

Conservationの構文 (1)

```
DEFINITION =
      define VARS -> NAME(VARS) -> VARS    S end
S = S … S                                          順序実行
|     cond WHEN… end                               条件分岐
|     VARS -> NAME(EXPS) -> VARS;                  関数呼び出し
|     VARS -> !NAME(EXPS) -> VARS;                 関数呼び出し
WHEN = when { S } { S } { S }
VARS = VAR, … VAR
EXPS = EXP, … EXP
```

Conservationの構文 (2)

EXP = LITERAL
| VAR
| NAME(EXP, ...)
| EXP BINOP EXP
| ...

- EXP は単方向式
- 逆方向実行できなくてよい、普通の式（副作用なし）

逆方向実行可能関数の定義

```
DEFINITION =
    define VARS -> NAME(VARS) -> VARS
           S
    end
```

- 変数がなければ "VARS ->" や "-> VARS" は省略可能
- 3種類の引数:
  - 順方向引数: 順方向では引数、逆方向では返値
  - 単方向引数: どちらの方向でも引数
  - 逆方向引数: 順方向では返値、逆方向では引数
- 文
  計算を行う本体
- 呼び出しごとに変数環境が作られる

順序実行

S = S … S

- 順方向: それぞれの文を左から右に順方向実行する
- 逆方向: それぞれの文を右から左に逆方向実行する

条件分岐

S = cond WHEN… end

WHEN = when { S } { S } { S }

- 各分岐は以下の3つの文からなる
  - 左ガード
  - 本体
  - 右ガード
- 順方向: まずすべての左ガードを順方向実行する。
  そのなかのひとつだけが成功したことを確認する。
  成功した分岐の本体と右ガードを順方向実行する。
  実行後の状態からすべての右ガードを逆方向実行する。
  成功した分岐の右ガードだけが成功し、他は失敗することを確認する。
- 確認に失敗して分岐を選べなければ失敗

関数呼び出し

```
S =     VARS -> NAME(EXPS) -> VARS;   同方向
|       VARS -> !NAME(EXPS) -> VARS;  反対方向
```

- 変数がなければ "VARS ->" や "-> VARS" は省略可能
- "!"の有無で関数を同方向に呼び出すか反対方向に呼び出すか選ぶ
- 左の変数: 同方向なら順方向引数、反対方向なら逆方向引数
- 右の変数: 同方向なら逆方向引数、反対方向なら順方向引数
- 順方向: 左の変数を消費して値を取り出す。
  式を評価して値を得る。
  呼び出される関数内の変数環境を作って本体を実行する。
  実行結果の変数環境から値を取り出して右の変数に束縛する。

bind 関数

bind(x) -> y

- 順方向引数はない
- 順方向: 単方向引数xの値を逆方向引数yに束縛
- 逆方向: 逆方向引数yを消費してその値が単方向引数xの値と等しいか比較
  異なっていたら失敗

guard 関数

guard(x)

- 順方向引数、逆方向引数はない
- 単方向引数xが真なら成功、偽なら失敗

divmod 関数

n -> divmod(m) -> q, r

- 順方向:
  q = floor(n/m)
  r = n – qm
  m が 0 なら失敗
- 逆方向:
  n = qm + r
  m が 0、r < 0, m ≦ r のいずれかが成り立てば失敗

cons 関数

h, t -> cons() -> l

- 順方向: 頭部 h, 尾部 t からリストを作って l に束縛
- 逆方向: consセル l を分解して頭部を h, 尾部を t に束縛
  l が consセルでなかったら失敗

自然数と文字列の変換

```
define n -> nat2str() -> s
  cond
  when { n -> !bind(0); }
    {}
    { bind([]) -> s; }
  when { guard(0 < n); }
    {
      n -> divmod(10) -> q, r;
      q -> nat2str() -> t;
    }
    { r, t -> cons() -> s; }
  end
end
```

- nは1以上の整数
- sは整数(0〜9)のリスト
- n -> !bind(0)
  順方向: n が 0 という表明
- bind([]) -> s
  順方向: s に空リストを束縛

X.509 証明書のファジング

- ファジング: 間違った入力を与えて変なことが起こらないかどうかテストする
- 証明書のエンコーダ・デコーダを利用して間違った証明書を生成する
- 暗号ライブラリで間違った証明書の署名を検証する
- 間違った証明書で暗号通信路を確立する
- 具体的には OpenSSL をテストする

間違った証明書の生成

- 正しい証明書をデコードする
- 証明書をエンコードするときに間違いを仕込む
  - 適切なところで変数の値を変える

Conservationの構文 (3)

S = backward_fuzzing(FUZZNAME) { FUZZ }

backward_fuzzing(FUZZNAME) { FUZZ } if FUZZCOND

- 逆方向実行のときにファジングする
  （エンコーダを逆方向として記述したため）
- 順方向: なにもしない
- 逆方向: FUZZCOND を評価して適用可能かどうか判断する。結果が偽なら適用できない。"if FUZZCOND" がなければ真の扱い。真でありかつ外部の設定により指示されれば適用する。
  適用されるときには FUZZ により変数を変更する
  FUZZNAMEは外部の設定でファジングを選ぶために使われる

ファジング後の動作

ファジングで変数が書き換えられた後:

- 条件分岐の表明は検査しない
  （順方向の右ガード、逆方向の左ガード）
- 条件分岐の条件以外での guard 関数呼び出しは常に成功する

ファジング例: 予期しない入力の終了

- 証明書はバイトのリスト
- デコーダはリストを分解する
- エンコーダはリストを構成する
- エンコーダの中で、構成している途中のリストを空リストに置き換える
  （cons() で構成したリストを空リストで置き換える）
  途中で途切れたリストが結果になる

ファジング例: 長すぎる整数エンコーディング

- 証明書はさまざまな整数を含む:
  タグ番号、内容の長さ、など
- 整数はもっとも短い表現でなければならない
  (ASN.1 DER の制約)
- ファジングで整数を冗長に表現する

実装したファジング

8種類実装した

- premature_end_at_content
- premature_end_at_hi_tag_number_octet
- premature_end_at_content_length_beginning_octet
- premature_end_at_long_content_length
- too_long_tagnum_encoding
- too_long_length_encoding
- too_long_length_encoding_max
- too_long_integer_encoding

どれも確実に間違った結果を生成する

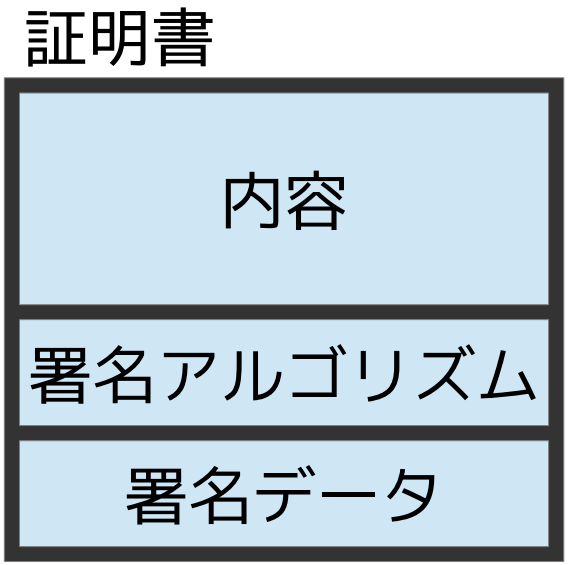
実験

- ASN.1 DER のエンコーダ・デコーダを実装
- 認証局証明書とサーバ証明書を正しく生成
- サーバ証明書をデコード
- デコード結果を間違ってエンコード
  - 1回間違った証明書を生成（404種類）
  - 2回間違った証明書を生成（82405種類）
- 間違った証明書の署名検証
- 間違った証明書で暗号通信路の確立に挑戦

結果

- テストは異常終了しなかった
- 暗号通信路は確立しなかった
- 署名検証が成功する場合があった

  署名の対象となる内容以外で冗長な整数表現が使われる場合
  内容の長さなど



まとめと今後の課題

- エンコーダ・デコーダを実装できる逆方向実行可能言語Conservationを提案した
- ファジングにも利用できることを示した
  - 記述したエンコーダ・デコーダは ASN.1 DER
  - 証明書固有の構文を利用したい
  - ベースとなる証明書を使わずにスクラッチから証明書を生成したい
- その他の課題
  - 実装をまともにして公開したい

逆方向実行の応用の可能性

- エンコーダ・デコーダ
- エディタなどのUNDO
- データベースのrollback

mknode 関数

v, … -> mknode(n, c) -> t

- 順方向: 木構造のノードの生成
  非終端記号n, 導出規則c のノード
  子ノードのとして v, ... の値をもつ
  t に束縛
- 逆方向: 木構造のノードを分解
  t が非終端記号n,導出規則c のノードか確認
  子ノードを v, ... に束縛