F*ハンズオン

FWS 運営委員会

Contents

1　はじめに

本資料は，コンピュータセキュリティシンポジウム 2024 の形式検証とセキュリティワークショップ（FWS）[1] 内のハンズオン企画「F*ハンズオン」の資料になります．

1.1　ハンズオンの概要

F* [2] はプログラム検証を目的に開発された関数型プログラミング言語です．型システムを用いてプログラムの正しさや安全性を検証することができ，セキュリティ分野にも多数応用されています．

　このハンズオンでは，F*の基本的な使い方を習得し，簡単なセキュリティ分野の応用例を理解することをゴールとします．具体的には，

- F*の文法
- F*によるプログラミング
- F*によるプログラムの検証

の基礎を理解してもらうことをゴールとします．より詳細については F*公式ホームページ [RI24] をご参照ください．なお，このハンズオンの内容はこちらのテキスト [Tea24, SMR23] を基に作成しています．

　本テキストで扱うコードおよび参考解答は FWS ハンズオンのページ

- https://www.iwsec.org/fws/2024/hands-on.html

から入手可能です．見やすさの観点からコードはテキスト形式で置いてありますが，実際の F*のコードの拡張子は.fst になります．

1.2　オンラインデモの使い方

このハンズオンでは F*のオンラインブック [Res24] にあるオンラインデモを使用します．

- https://fstar-lang.org/tutorial

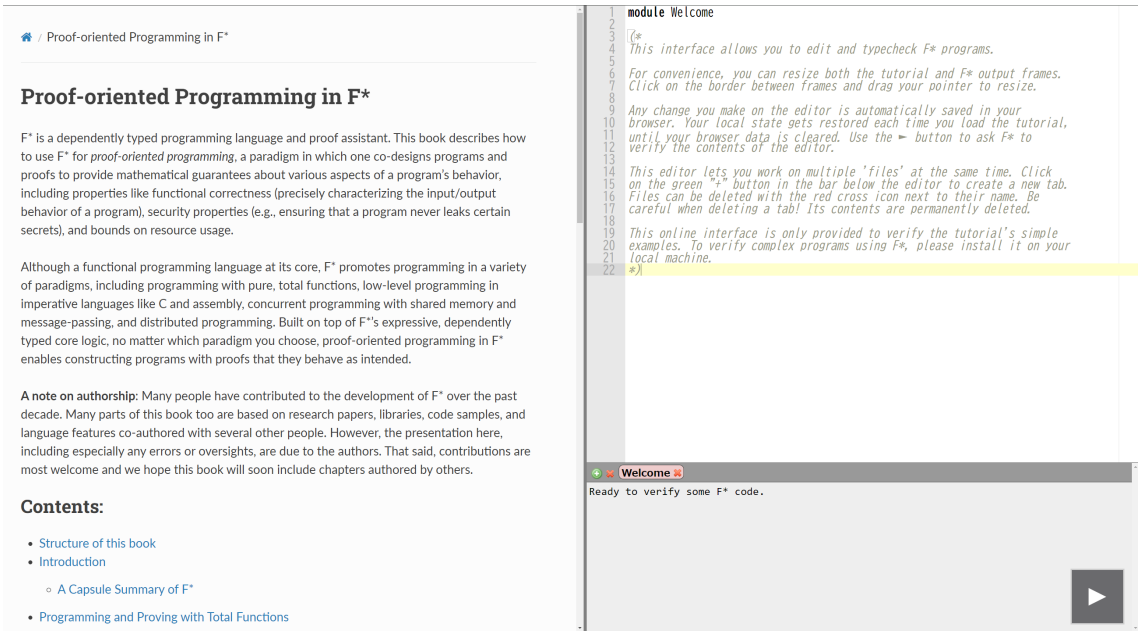

図 1: F*オンラインデモの画面．

[1] https://www.iwsec.org/fws/2024/

[2] F*という名前は，System F から来ています．Fable，F7，F9，F5，FX，F#などなど System F を起源とする F なんちゃら系言語もたくさんあるようです．*は fixpoint operator から来ていたり，分離論理の*の意味合いもあるようです．

右側のフォームにコードを入力し，右下の実行ボタン（▷ マーク）を押すとプログラム検証が実行され，右下の領域に出力が表示されます．オンラインデモでも一部のライブラリを読み込んで使うことができます．

なお，F*はデフォルトでは入力されたコードに対して検証のみを行います．実行はされません．F*のコードを実行するためには，OCaml や F#のコンパイラを用いてコンパイルする必要があります．F*のコードの実行についてはこちらをご参照ください [Hri24].

1.3 型システムとは

型システムとは，プログラムの各部分を，プログラムが計算する値の種類に沿って分類することにより，プログラムがある種の振る舞いを起こさないことを保証する形式手法の一種です [Pie13]．型システムによる検証を型検査と呼び，プログラムが型検査で安全（型安全）であるならばそのプログラムの中にある種のエラーが存在しない，と結論づけることができます.「ある種の」と書きましたが，検出できるエラーは型システムの種類に依存します．本ハンズオンで対象としている F*も型システムを用いてプログラムの正しさや安全性を検証しています.

2 基礎編：F*を動かしてみよう

2.1 基礎文法

はじめに F*の基礎文法について説明します．文法の詳細については [SMR23] などをご覧ください．このハンズオンでは時間の都合上，おまじない部分が多くなってしまいますがどうぞご了承ください.

F*のプログラムはいくつかのモジュールで構成され，モジュールは読み込むライブラリ（モジュール）のリスト，シグネチャ，定義のリストから構成されます.

```
module Sample1
  open Lib
  type ...
  let ...
  val ...
```

コメント表現は//または (* ... *) です.

シグネチャは定義に型を割り当てるもので，val から始まります（val f:t など）．定義にはいくつかの種類があり，let で始まる再帰的な定義（let [rec] f = e）や type で始まる帰納的な型の定義（例えば type t = | D1:t1 | ... | Dn:tn）があります.

F*ではリファインメント型を用いた定義も可能です．リファインメント型は型の要素に対して成立する述語が付与された型で，変数 x，型 t，述語 e に対して x:t{e} の形式で定義されます．例えば，自然数の型 nat は次のように表現できます.

```
let nat = x:int{x >= 0}
```

ブール演算や int 型に対する各種演算は次のように定義されています.

```
  true
  false
  not // 否定
  && // かつ
  || // または

  x - y // 引き算
  x + y // 足し算
  x / y // 割り算
  x % y // 剰余
  x < y // 比較
  x <= y
```

int 型に対する各種演算-，+，/，%，<，<=，... も定義されています.

条件分岐（if 文）は次のように記述します．condition には条件式が入ります.

```
  if condition then ...
  else ...
```

関数の定義は次のように記述します．

```
let func (x:type) = ...
let func = fun (x:type) -> ...
```

上記の 2 行はどちらも同じ意味を持ちます．例えば int 型の 2 つの変数 x，y を引数に取り，x > y ならば true を返す関数 more_than は次のように記述できます．

```
let more_than (x:int) (y:int): bool = x > y
```

パターンマッチによる定義も可能です．例えば，x = a ならば true を，それ以外ならば false を返す関数 match_a はパターンマッチ（match 文）を用いて次のように記述できます．

```
let match_a (x:type) =
  match x with
    | a -> true
    | b -> false
```

演習 2.1. 次の型と関数を定義してみましょう．(TypesAndFuncs.fst)

- （int 型を用いて）偶数の型
- 0 以上の int 型の 2 つの値を受け取り，それらの和を返す関数
- （if 文を用いて）int 型の値を受け取り，0 未満の値なら true を，それ以外ならば false を返す関数

2.2 最初の例：アクセスコントロールのモデル

次に，最初の例としてシステムのアクセスコントロールのモデルを考えてみます．この章は [Tea24] の 1.1 節を基に作成しました．

この例では，ファイルの読み書きのアクセスコントロールモデルを考えます．各ファイルに対するアクセスポリシーが定義されているときに，それが正しく実装されていることを検証することを目標とします．流れは次の通りです．

1. アクセスポリシーを定義する．
2. ファイルアクセスに関するプリミティブを定義する．
3. 安全性を検証する．

まずアクセスコントロールモデルに対するアクセスポリシーを定義し，次にそのアクセスポリシーに基づきファイルの読み書きを行うプログラムを作ります．そして，プログラムがアクセスポリシーに基づいて正しく実装されていることを検証します．

2.2.1 アクセスポリシーを定義する

次のプログラムを考えてみましょう．

コード 1: アクセスポリシーのリスト（ACLs.fst）

```
module ACLs
  type filename = string

  let canWrite (f:filename) =
    match f with
      | "demo/tempfile" -> true
      | _ -> false

  let canRead (f:filename) =
    canWrite f
    || f="demo/README"
```

モジュール ACLs（Access-Control Lists）ではアクセスポリシーを定義しています．なお，ここではアクセスポリシーはユーザ毎にアクセス可否の権限を与えるようなものではなく，ファイル毎に付与されているものと仮定しています．canWrite はファイル f に書き込みが可能かどうかを判断する関数で，パターンマッチを用いて f = "demo/tempfile"であれば true（書き込み可能）を，そうでなければ false（書き込み不可能）を返します．canRead はファイル f が読み出し可能かどうかを判断する関数で，同様にパターンマッチを用いて f が書き込み可能または"demo/README"であるならば読み出し可能になります．

まとめると，ここで定義しているアクセスポリシーは

- ファイル名が demo/tempfile であれば書き込み可能
- ファイル名が demo/tempfile または demo/README であれば読み込み可能

になります．

2.2.2 ファイルアクセスプリミティブを定義する

次にファイルの読み書きを行うプリミティブを定義します．F*では，別の言語等で実装された外部モジュールのインターフェイスを指定する機能をサポートしています．例えば，OS で実装されているファイルの入出力操作に対し，.NET や OCaml のような下層のフレームワークを通して F*プログラム中で有効にすることができます．そのフレームワークを用いることによって，ファイルの読み書きは次のようなモジュールで表現できます．

コード 2: ファイルアクセスプリミティブ（FileIO.fst）

```
1 module FileIO
2   open ACLs
3   assume val read : f:filename{canRead f} -> string
4   assume val write : f:filename{canWrite f} -> string -> unit
```

ここでは，先ほど定義した ACLs モジュールを用いて FileIO という 2 つの関数 read，write を備えたモジュールを定義しています[3]．read 関数は引数として canRead f が true と評価される（すなわち読み出し可能な）ファイル名を取り，string 型の値（読み出した値）を返します．write 関数は引数として canWrite f が true と評価される（すなわち書き込み可能な）ファイル名と string 型の変数（書き込む値）を取り，unit 型の値を返します．ここで，unit 型は唯一の要素 () を持つ型で，C 言語の void 型のようなものです．

2.2.3 安全性を検証する

次に，プログラムが 2.2.1節で定義したアクセスポリシーに従っていることを検証してみましょう．次のプログラムを考えてみます．

コード 3: クライアントコードの例（ClientCode.fst）

```
1 module ClientCode
2   open FileIO
3   let passwd = "demo/password"
4   let readme = "demo/README"
5   let tmp = "demo/tempfile"
6
7   let staticChecking () =
8     let v1 = read tmp in
9     let v2 = read readme in
10    write tmp "hello!"
```

このプログラムでは，上で定義した FileIO モジュールを用いて複数のファイルの読み書きを行う関数 staticChecking を定義します．8 行目はファイル demo/tempfile からの読み出しを，9 行目はファイル demo/README からの読み出しを意味し，読み出した結果をそれぞれ v1 と v2 に格納します．10 行目ではファイル demo/tempfile に hello!という文字列を書き込みます．

[3]F*では検証の際に SMT ソルバを用います．コード 2の 3 行目と 4 行目の一番左についている assume は SMT ソルバに仮定として伝える命題であることを意味しています．

演習 2.2. コード 1から 3を連結させてオンラインデモに入力し，出力結果を確認してみましょう．

次のような結果が出力されたと思います．

```
Verified module: ACLs
Verified module: FileIO
Verified module: ClientCode
All verification conditions discharged successfully
```

各モジュールごとに型検査が行われ，その結果が出力されています．このプログラムでは読み出し可能なファイルへの読み出しと書き込み可能なファイルへの書き込みしか行っていないため，全てのモジュールに対して型検査が成功します．この例では，

$$\text{型検査が成功する（型安全）} \Leftrightarrow \text{プログラムがアクセスポリシーを満たす}$$

となります．

演習 2.3. 演習 2.2のプログラムに読み出し不可能なファイルへのアクセスを追加し，出力結果を確認してみましょう．

参考解答は FWS の Web ページにある演習 2.3 参考解答（ClientCodeNG.fst）をご覧ください．以下では参考解答の出力を想定して話を進めていきます．ClientCodeNG.fst の出力は次のようになります．

```
* Error 19 at ClientCode.fst(10,18-10,24):
  - Subtyping check failed
  - Expected type f: FileName.filename{ACLs.canRead f} got type Prims.string
  - The SMT solver could not prove the query. Use --query_stats for more details.
  - See also FileIO.fst(3,32-3,41)

Verified module: FileName
Verified module: ACLs
Verified module: FileIO
Verified module: UlientCode
1 error was reported (see above)
```

`ClientCode` モジュールにおいてエラーが見つかっています．これは，読み出しが許可されていないファイル `demo/password` を読み出しているために起きているエラーです．このようにして，アクセスコントロールモデルが正しく実装されているか，すなわち許可された操作のみを可能としているかを検査することができます．

通常のプログラミングでは，（ソースコードを注意深く見ない限り）プログラムを実行してテストしてからでないとアクセス権のないファイルにアクセスしていることに気づくことはできません．型システムを用いることで，実行前にエラーに気づくことができます．

3　応用編：セキュリティへの応用

続いて，暗号実装への応用として，マークルツリー（Merkle Tree）の例を考えてみます．マークルツリーは Ralph Merkle によって導入されたデータ構造で，ハッシュ関数を用いてツリーに格納されているデータの真正性を効率的に証明することができます．

ここではシンプルなマークルツリーを作り，その安全性を示します．この章は [SMR23] の 14 章を基に作成しました．

3.1　マークルツリーとは

マークルツリーは高さ n の完全二分木で，2^n 個のデータ項目とそれらに対応するハッシュ値をノードに格納します．マークルツリーでは，葉ノードにデータのハッシュ値が格納され，各内部ノードはその子ノードに格納されているハッシュ値のハッシュ値を保持します（図 2参照）．

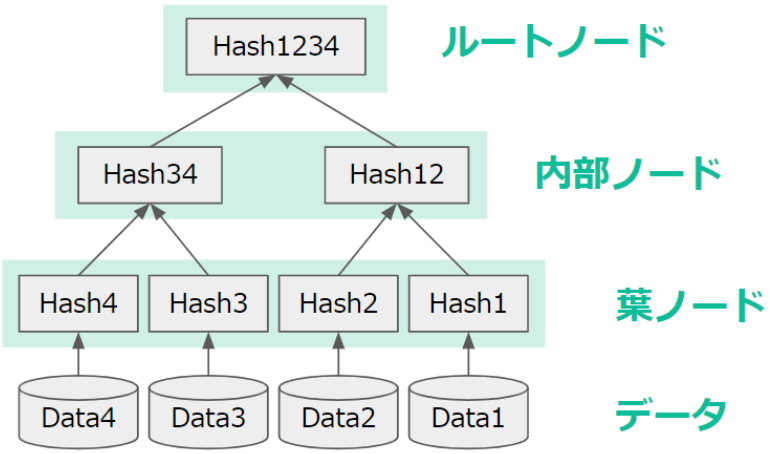

図 2: マークルツリーの概要図.

使用されるハッシュアルゴリズムが衝突困難性[4]を満たす場合，ルートノードに関連付けられたハッシュ値によってツリーに含まれるデータ全体が認証されます（すなわち，ルートノードが正しい場合，すべてのデータ項目は改ざんされていないことが証明できます）．さらに，マークルツリーの中に特定のデータが存在することを，ルートからそのデータを含む葉までのパス内の兄弟ノードに格納されているハッシュ値のリストのみから示すことができます．

3.2　マークルツリーのモデルとその検証

以下では，マークルツリーと周辺の機能をモデリングし，その安全性を検証していきます．手順は次の通りです．

1. マークルツリーを定義する．
2. ツリーの各データへのアクセス関数を定義する．
3. 証拠付きツリーの各データへのアクセス関数を定義する．
4. 証拠を検証する関数を定義する．
5. 安全性を証明する．

まずマークルツリーを定義し，その後にデータ ID という識別子を用いてツリーの各データにアクセスする関数を定義します．続いてその関数を拡張し，ツリーの中にそのデータが存在するという証拠も一緒に提供するアクセス関数を定義します．そして証拠が正しいことを検証する関数を定義し，最後にここまでの定義がマークルツリーの安全性（ハッシュ関数が衝突困難ならば不正な証拠を生成できない）を満たすことを証明します．

[4]ハッシュアルゴリズムの安全性の一つで，ハッシュ値が一致するような異なる入力値のペアを見つけることは計算量的に難しい，という性質のことです．

3.2.1 マークルツリーを定義する

では，マークルツリーを定義していきましょう．まずは，ツリーに格納するデータとハッシュ値を文字列型でモデリングします．ここでは文字列操作に関するライブラリ FStar.String からいくつかの関数を読み込んでいます．

コード 5: マークルツリーの定義（MerkleTree.fst）1/2

```
1  let lstring (n:nat) = s:string{String.length s == n}
2
3  let concat #n #m (s0:lstring n) (s1:lstring m)
4      : lstring (m + n)
5      = FStar.String.concat_length s0 s1;
6          s0 ^ s1
7
8  assume val hash_size:nat
9
10 let hash_t = lstring hash_size
11
12 assume val hash (m:string) : hash_t
13
14 let mtdata = string
```

lstring は長さで特徴付けられた文字列の型で，文字列の長さに関する関数 String.length を用いて長さが n の文字列の型を定義しています．concat は長さ n と m の文字列を連結させた，長さ n+m の文字列の型です[5]．hash_size はハッシュ関数の出力文字列の長さを表すパラメータであり，hash 関数は文字列 m を受け取ってハッシュ値の型 hash_t の文字列を返すハッシュ関数です．型 mtdata はツリーに格納するデータの型です．

それではマークルツリーの型 mtree を定義しましょう．

コード 5: マークルツリーの定義（MerkleTree.fst）2/2

```
1  type mtree: nat -> hash_t -> Type =
2  | L:
3      data:mtdata ->
4      mtree 0 (hash data)
5
6  | N:
7      #n:nat ->
8      #hl:hash_t ->
9      #hr:hash_t ->
10     left:mtree n hl ->
11     right:mtree n hr ->
12     mtree (n + 1) (hash (concat hl hr))
```

型 mtree は 2 つのインデックスを持ち，型 mtree n h はルートノードがハッシュ値 h に関連付けられている高さ n のマークルツリーの型を意味します．

2 行目から 4 行目ではツリーの葉ノード（L）を定義しています．葉ノードは高さ 0 のツリーで，ノードにはデータの値 data が格納されています．6 行目から 12 行目は内部ノード（葉以外のノード）を定義しており，ノード N left right は左側に left:mtree，右側に right:mtree という高さ n のツリーを持つノードを表現しています．ノード N left right に格納されているハッシュ値は左右のサブツリーのハッシュ値の連結のハッシュ値 hash (concat hl hr) になっています．

3.2.2 ツリーの各データへのアクセス関数を定義する

ツリーの各データにアクセスする機能を実装しましょう．ツリーの各データへのアクセスは，ルートからそのデータを格納する葉ノードまでのパス（各ノードから左に降りるか右に降りるかを示すブール値のリスト）を指定することで行うことができます．このパスをデータ ID と呼ぶことにします．

マークルツリー内の各データにアクセスする関数 get は次のように定義できます[6]．

コード 6: データアクセス関数の定義（MTAccess.fst）

[5] #が付いている変数は implicit argument と呼ばれるもので，自動的に推論可能な場合などで省略可能な変数を意味します．

[6] Tot というラベルは関数が total function であるという注釈を付与しています．decreases did は再帰呼び出しの中でリスト did が確実に減少することを意味し，プログラムが停止することを保証するために必要になります．

```
1  let data_id = list bool
2
3  let rec get #h
4          (did:data_id)
5          (tree:mtree (Lib.length did) h)
6      : Tot mtdata (decreases did)
7      = match did with
8      | [] -> L?.data tree
9      | b::did' ->
10         if b then
11             get did' (N?.left tree)
12         else
13             get did' (N?.right tree)
```

data_id はデータ ID の型で，ブール値のリストとして定義されています．Lib.～の形で書かれる関数はライブラリ[7]から読み込んでいる関数で，Lib.length はリストの長さを測る関数です．関数 get では，データ ID を頭から読み込み，パターンマッチを用いてルートノードから順に左に降りるか（10-11 行目）右に降りるか（12-13 行目）を決定し，リソース ID を最後まで読み切ると葉ノードのデータの値を返します（8 行目）．文字列上の match 文では，文字列を 1 文字ずつ読み込み，その結果に応じた動作を定義できます．この例では，ブール値のリスト did に対して，

- did が空であれば
  - L?.data tree を返す
- did が b::did' の形（1 つ目の要素がブール値 b，それ以降がブール値のリスト did'）であれば
  - もし b = true ならば get did' (N?.right tree) を返す
  - そうでなければ get did' (N?.right tree) を返す

というように再帰的に定義されています．

演習 3.1. get 関数を用いて，ルートノードが h であるマークルツリー tree のデータ ID did が指定するデータを表現してみましょう．

答えは get h did tree です．ルートノードのハッシュ値，マークルツリー，データ ID の順で get 関数に引数を渡すことでデータ ID が指し示すデータを指定することができます．

3.2.3 証拠付きのツリーの各データへのアクセス関数を定義する

マークルツリーでは，特定のデータがツリー内に存在する証拠を提供できます．ここでは証拠はデータ ID とルートからそのアイテムまでのパスに沿った兄弟ノードのハッシュから構成されるものとします（図 3参照）．

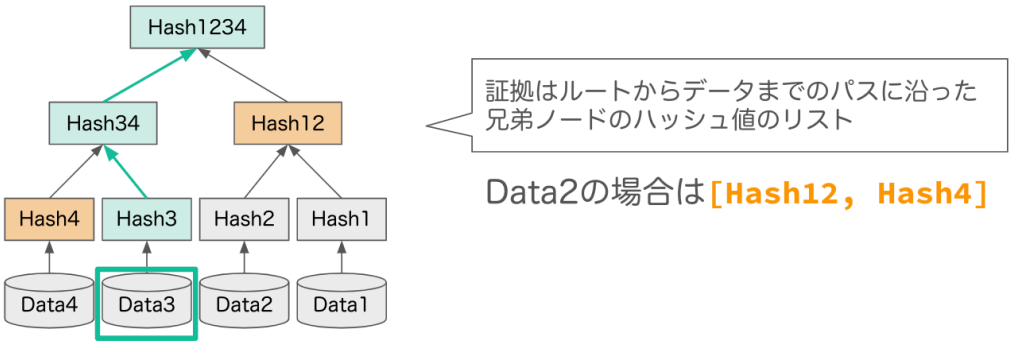

図 3: マークルツリーの証拠．

[7] ここでは，リスト操作に関するライブラリ Fstar.List を Lib という名前で読み込んでいます．

先ほど定義した関数 get に，証拠も同時に提供する機能を付け加えた関数 get_with_evidence を定義しましょう．そのために，まずインデックス付きの型 data_with_evidence を定義します．これはデータをそのデータ ID とハッシュ値のリストでパッケージングするものです．

コード 7: 証拠付きデータアクセス関数の定義（MTAccessEvidence.fst）1/2

```
type data_with_evidence : nat -> Type =
    | DATA:
        data:mtdata ->
        did:data_id ->
        hashes:list hash_t { Lib.length ri == Lib.length hashes } ->
        data_with_evidence (Lib.length ri)
```

証拠付きデータ p:data_with_evidence に対し，それぞれの要素は

- データ：p.data
- データ ID：p.did
- ハッシュ値のリスト：p.hashes

と記述することで指定可能です．

演習 3.2. 次のコードの ??? 部分を埋めて，証拠付きデータを返す関数 get_with_evidence の型を定義してみましょう．引数はマークルツリーのルートノード h，データ ID did，マークルツリー tree で，証拠付きデータを返します．
基本的な動作は関数 get と同様で，ノードを上から辿っていくときに該当するハッシュ値をリストに追加していき，最後にそのリストを返すような関数になっています．

コード 7: 証拠付きデータアクセス関数の定義（MTAccessEvidence.fst）2/2

```
let rec get_with_evidence (#h:_) ??? ???
    : Tot ??? (decreases did)
    = match did with
    | [] ->
        DATA (L?.data tree) [] []

    | b::did' ->
        let N #_ #hl #hr left right = tree in
        let p = get_with_evidence did' left in
        if b then
            let p = get_with_evidence did' left in
            DATA p.data did (hr :: p.hashes)
        else
            let p = get_with_evidence did' right in
            DATA p.data did (hl :: p.hashes)
```

3.2.4 証拠を検証する関数を定義する

証拠を受け取り，提示された証拠からルートハッシュを再計算し，そのハッシュが指定されたマークルツリーのルートハッシュと一致するかどうかを確認する関数 verify を実装します．

そのために，まず証拠からルートハッシュを計算する関数 compute_root_hash を考えます．関数 compute_root_hash では，

- 最初の分岐ではデータ自体をハッシュし，
- 2 番目以降の分岐ではデータ ID の末尾からハッシュを再計算し，どの方向を取ったかに基づいて，左側または右側の兄弟ハッシュを連結し，そのハッシュ値を計算します．

コード 8: 証拠を検証する関数の定義（VerifyEvidence.fst）の一部

```
let tail #n (p:data_with_evidence n { n > 0 })
    : data_with_evidence (n - 1)
```

```
    = DATA p.data (Lib.tail p.did) (Lib.tail p.hashes)

let rec compute_root_hash (#n:nat)
                          (p:data_with_evidence n)
    : hash_t
    = let DATA d did hashes = p in
    match did with
    | [] -> hash p.data
    | b::did' ->
        let h' = compute_root_hash (tail p) in
        if b then
            hash (concat h' (Lib.hd hashes))
        else
            hash (concat (Lib.hd hashes) h')
```

ここで，関数 tail は長さ n の証拠付きデータからそれぞれの要素の背後部分を抽出した長さ n-1 の証拠付きデータを返す関数です．

演習 3.3. 次のコードの ??? 部分を埋めて，関数 compute_root_hash を用いて，証拠とハッシュ値を受け取り，証拠から計算されたルートハッシュとそのハッシュ値が一致すれば true を，そうでなければ false を返す関数 verify を定義してみましょう．なお，値の一致は=で表現することができます．

コード 8: 証拠を検証する関数の定義（VerifyEvidence.fst）の一部

```
let verify #h #n (p:data_with_evidence n) (tree:mtree n h)
    : bool
    = ???
```

F*では，定理の形で性質を示すこともできます．次の定理 correctness を考えることで，get_with_evidence の正しさ，つまり get_with_evidence がいつも正しいエビデンス（verify 関数で true と評価される）を返すことを確認できます．

コード 8: 証拠を検証する関数の定義（VerifyEvidence.fst）の一部

```
let rec correctness (#h:hash_t)
                    (did:data_id)
                    (tree:mtree (Lib.length did) h)
    : Lemma (ensures (verify (get_with_evidence did tree) tree))
            (decreases did)
    = match did with
    | [] -> ()
    | b::did' ->
        let N left right = tree in
        if b then
            correctness did' left
        else
            correctness did' right
```

3.2.5 安全性を証明する

最後にマークルツリーの安全性を考えてみます．ここで考えるマークルツリーの安全性は，直感的には「ハッシュ関数が衝突困難ならば不正な証拠を生成できない」ということです．そのために「マークルツリーの中に存在しないデータの証拠が検証者によって受け入れられる場合，ハッシュ関数の衝突を構築できること」を示します．具体的には，ハッシュ値が同じ値になるような 2 つの異なる文字列のペアの型を定義し，不正な証拠が検証で受け入れられる場合はその型の値を返すような関数を定義できることを示すことで安全性を示します．

はじめに，型 hash_collision を定義しましょう．これはハッシュ値が同じ値になるような 2 つの異なる文字列のペアの型です．

コード 9: 安全性の証明（ProofSecurity.fst）の一部

```
type hash_collision =
```

```
2     | Collision :
3         s1:string ->
4         s2:string {hash s1 = hash s2 /\ not (s1 = s2)} ->
5         hash_collision
```

そして安全性を表現する次の関数 security を考えます.

コード 9: 安全性の証明（ProofSecurity.fst）の一部

```
1  let rec security (#n:nat) (#h:hash_t)
2                       (tree:mtree n h)
3                       (p:data_with_evidence n {
4                           verify p tree /\
5                           not (get p.did tree = p.data)
6                       })
7      : hash_collision
8      = match p.did with
9      | [] -> Collision p.data (L?.data tree)
10     | b::did' ->
11         let N #_ #h1 #h2 left right = tree in
12         let h' = compute_root_hash (tail p) in
13         let hd :: _ = p.hashes in
14             if b then
15                 if h' = h1 then
16                     security left (tail p)
17                 else (
18                     String.concat_injective h1 h' h2 hd;
19                     Collision (concat h1 h2) (concat h' hd)
20                 )
21             else
22                 if h' = h2 then
23                     security right (tail p)
24                 else (
25                     String.concat_injective h1 hd h2 h';
26                     Collision (concat h1 h2) (concat hd h')
27                 )
```

この関数をコンパイルしてエラーが起きないときに「マークルツリーの中に存在しないデータの証拠が検証者によって受け入れられる場合ハッシュ関数の衝突を構築できる」ことを示せるため，ここまで定義してきたマークルツリーに関する定義が安全性を満たすことがわかります.

それぞれの場合分けを見てみましょう．ベーシックケース（9 行目）では，異なるデータから直接ハッシュの衝突を構築できます．それ以外の場合（10-27 行目）は，提示された証拠の末尾から現在のノードに関連付けられたハッシュを左右のツリー分再計算します．もし再計算されたハッシュがノードのハッシュと一致する場合（15-16 行目と 22-23 行目），左または右のサブツリーの帰納法の仮定を用いて衝突を構築できます．それ以外の場合（17-20 行目と 24-27 行目）は，ライブラリの補題である String.concat_injective を利用してハッシュ衝突を構築できます．この補題は，長さが等しい 2 組の文字列の連結は，そのコンポーネントが等しい場合にのみ等しいという補題です．h' <> h1 (または h' <> h2，記号<>は $\neq$) であることが分かっているため、連結が等しくないことを証明できますが，ハッシュは仮定により等しいとされているため，衝突を構築することができます.

4 おわりに

F*ハンズオンにご参加いただきありがとうございました．本ハンズオンでは，特にセキュリティ分野への応用にフォーカスして F*によるプログラミングとその検証の基礎についてご紹介しました.

時間の都合上，おまじない部分が多くなってしまいましたが，検証やセキュリティへの応用方法の雰囲気を感じてもらえていたら嬉しいです．興味を持っていただけた方はぜひマニュアルや既存のプログラムなどを見ていただければと思います.

A　F*のセキュリティでの応用

最後に付録として，F*公式ページ [RI24] で紹介されている内容を基に，F*のセキュリティ分野での応用研究を紹介します．関連論文の完全版はこちらをご覧ください [IR24].

A.1　暗号実装ライブラリ

F*の暗号実装用ライブラリも多数存在します.

- HACL* [IRC19b]：C 言語向けの暗号プリミティブライブラリ.
- ValeCrypt [Cor24]：検証済みアセンブリ言語プログラミングフレームワーク Vale の暗号プリミティブライブラリ.
- EverCrypt [IRC19a]：HACL*と ValeCrypt を統合したフレームワーク.

これらは Mozilla Firefox [Beu17]，Linux カーネル [ICD19]，Python [msp22] などのプロジェクトで使われているそうです.

A.2　応用研究

いくつかの著名な論文を紹介します.

- Implementing and Proving the TLS 1.3 Record Layer（S&P 2017） [DLFK+17] ：Low*（C 言語にコンパイル可能な F*のサブセット）を用いて TLS 1.3 を実装.
- HACL*: A Verified Modern Cryptographic Library（CCS 2017） [ZBPB17] ：Low*で検証済みの暗号ライブラリを実装.
- Formally Verified Cryptographic Web Applications in WebAssembly（S&P 2019） [PBMB19] ：HACL*を用いて Signal プロトコルを実装.
- EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider（S&P 2020） [PPF+20] ：HACL*と ValeCrypt の C 言語とアセンブリコードを組み合わせた暗号化プロバイダを提案.
- HACL × N: Verified Generic SIMD Crypto (for all your favorite platforms)（CCS 2020）[PBP+20]：複数のアーキテクチャ向けに最適化された検証済み暗号ライブラリを構築するための F*を用いた手法を提案.
- A Security Model and Fully Verified Implementation for the IETF QUIC Record Layer（S&P 2021） [DLFP+21] ：Low*を用いて QUIC を実装.
- DICE*: A Formally Verified Implementation of DICE Measured Boot（USENIX Security 2021）[TRG+21] ：EverCrypt を用いて Measured boot プロトコル DICE を実装.
- DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code（Euro S&P 2021） [BBD+21a] ：暗号プロトコル実装の型ベースのシンボリック安全性検証フレームワークを F*で開発.
- An In-Depth Symbolic Security Analysis of the ACME Standard（CCS 2021） [BBD+21b] ：DY*を用いて ACME 証明書の発行・管理プロトコルの安全性を証明.
- Noise*: A Library of Verified High-Performance Secure Channel Protocol Implementations（S&P 2022） [HPBB22] ：F*を用いたセキュアチャネルプロトコルの検証済み C 言語実装生成フレームワークを提案.
- TreeSync: Authenticated Group Management for Messaging Layer Security（USENIX Security 2023） [WPBB23] ：DY*を用いて MLS を実装.
- Comparse: Provably Secure Formats for Cryptographic Protocols（CCS 2023） [WPB23] ：DY*を用いて暗号プロトコルのデータ形式に特化した検証フレームワークを提案.

References

[BBD+21a] Karthikeyan Bhargavan, Abhishek Bichhawat, Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, and Tim Würtele. Dy*: A modular symbolic verification framework for executable cryptographic protocol code. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 523–542. IEEE, 2021.

[BBD+21b] Karthikeyan Bhargavan, Abhishek Bichhawat, Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, and Tim Würtele. An in-depth symbolic security analysis of the acme standard. In *Proceedings of the 2021 ACM SIGSAC conference on computer and communications security*, pages 2601–2617, 2021.

[Beu17] Benjamin Beurdouche. Verified cryptography for firefox 57, 2017. https://blog.mozilla.org/security/2017/09/13/verified-cryptography-firefox-57/. Accessed: 2024-10-23.

[Cor24] Microsoft Corporation. Vale (verified assembly language for everest) cryptographic libraries, 2024. https://github.com/hacl-star/hacl-star/tree/main/vale. Accessed: 2024-10-23.

[DLFK+17] Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy, Santiago Zanella-Béguelin, Karthikeyan Bhargavan, Jianyang Pan, and Jean Karim Zinzindohoue. Implementing and proving the tls 1.3 record layer. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 463–482. IEEE, 2017.

[DLFP+21] Antoine Delignat-Lavaud, Cédric Fournet, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Jay Bosamiya, Joseph Lallemand, Itsaka Rakotonirina, and Yi Zhou. A security model and fully verified implementation for the ietf quic record layer. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1162–1178. IEEE, 2021.

[HPBB22] Son Ho, Jonathan Protzenko, Abhishek Bichhawat, and Karthikeyan Bhargavan. Noise: A library of verified high-performance secure channel protocol implementations. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 107–124. IEEE, 2022.

[Hri24] Catalin Hritcu. Executing f* code, 2024. https://github.com/FStarLang/FStar/wiki/Executing-F*-code. Accessed: 2024-10-23.

[ICD19] INRIA, Microsoft Corporation, and Jason A. Donenfeld. machine-generated formally verified implementation of curve25519, 2019. https://github.com/torvalds/linux/blob/0f2a4af27b649c13ba76431552fe49c60120d0f6/lib/crypto/curve25519-hacl64.c#L1. Accessed: 2024-10-23.

[IR24] INRIA and Microsoft Research. F* papers, 2024. https://fstar-lang.org/fstar_bib.html. Accessed: 2024-10-23.

[IRC19a] INRIA, Microsoft Research, and CMU. Evercrypt apis, 2019. https://hacl-star.github.io/EverCryptDoc.html. Accessed: 2024-10-23.

[IRC19b] INRIA, Microsoft Research, and CMU. A high assurance cryptographic library, 2019. https://hacl-star.github.io/. Accessed: 2024-10-23.

[msp22] msprotz. Replace built-in hashlib with verified implementations from hacl*, 2022. https://github.com/python/cpython/issues/99108. Accessed: 2024-10-23.

[PBMB19] Jonathan Protzenko, Benjamin Beurdouche, Denis Merigoux, and Karthikeyan Bhargavan. Formally verified cryptographic web applications in webassembly. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1256–1274. IEEE, 2019.

[PBP+20] Marina Polubelova, Karthikeyan Bhargavan, Jonathan Protzenko, Benjamin Beurdouche, Aymeric Fromherz, Natalia Kulatova, and Santiago Zanella-Béguelin. Haclxn: Verified generic simd crypto (for all your favourite platforms). In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 899–918, 2020.

[Pie13] Benjamin C Pierce. **型システム入門 プログラミング言語と型の理論**. 株式会社 オーム社, 2013.

[PPF+20] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, et al. Evercrypt: A fast, verified, cross-platform cryptographic provider. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 983–1002. IEEE, 2020.

[Res24] Microsoft Research. Proof-oriented programming in f*, 2024. https://fstar-lang.org/tutorial/. Accessed: 2024-10-23.

[RI24] Microsoft Research and INRIA. F*, 2024. https://fstar-lang.org/index.html. Accessed: 2024-10-23.

[SMR23] Nikhil Swamy, Guido Martinez, and Aseem Rastogi. Proof-oriented programming in f*, 2023.

[Tea24] The F* Team. F*における検証されたプログラミング チュートリアル, 2024. http://fstar-ja.metasepi.org/doc/tutorial. Accessed: 2024-10-23.

[TRG+21] Zhe Tao, Aseem Rastogi, Naman Gupta, Kapil Vaswani, and Aditya V Thakur. Dice*: A formally verified implementation of {DICE} measured boot. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1091–1107, 2021.

[WPB23] Théophile Wallez, Jonathan Protzenko, and Karthikeyan Bhargavan. Comparse: Provably secure formats for cryptographic protocols. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 564–578, 2023.

[WPBB23] Théophile Wallez, Jonathan Protzenko, Benjamin Beurdouche, and Karthikeyan Bhargavan. Treesync: authenticated group management for messaging layer security. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1217–1233, 2023.

[ZBPB17] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. Hacl*: A verified modern cryptographic library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1789–1806, 2017.