F*ハンズオン

CSS2024 形式検証とセキュリティワークショップ（FWS）

中林 美郷（NTT社会情報研究所） 花谷 嘉一（東芝） 米山 一樹（茨城大学）

はじめに

本ハンズオンでは，F*のオンラインデモを使います

インターネットにつながるPCをご用意ください

資料はこちら（FWSのページ）からダウンロードできます

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

はじめに

ご不明な点があればお気軽にどうぞ

- slackのFWSチャンネル
- 挙手による質問

目次

前半

後半

目次

ハンズオンの概要

F*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

プログラムが型安全 ⟺ プログラムがある種の振る舞いを起こさない

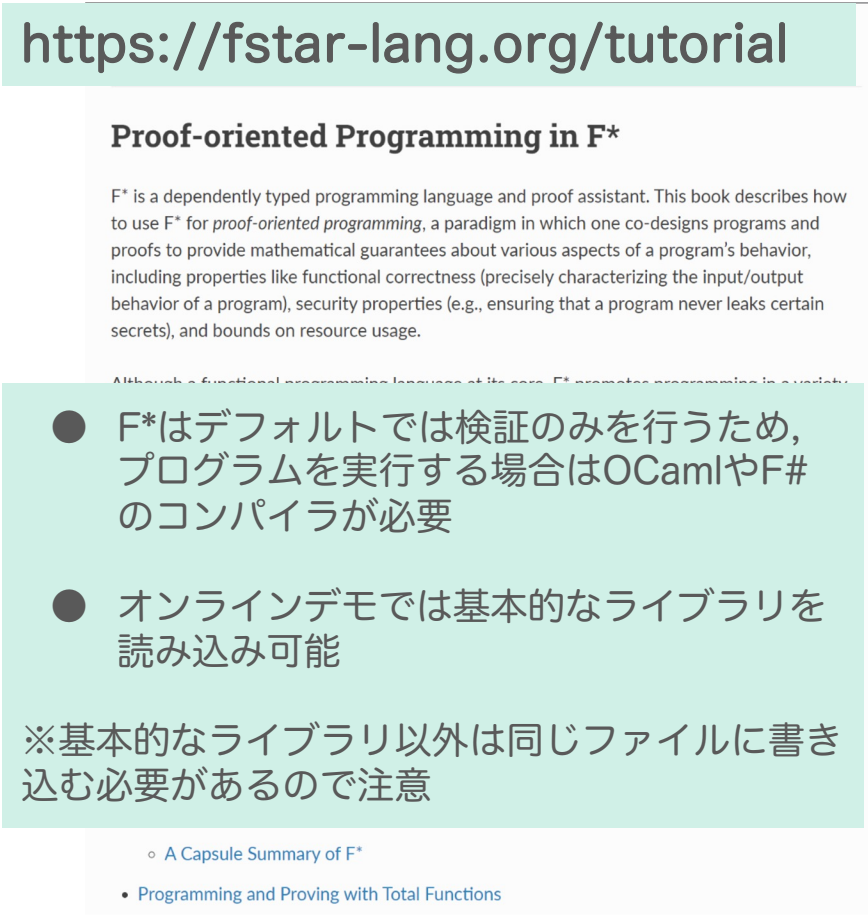
このハンズオンでは

- 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する

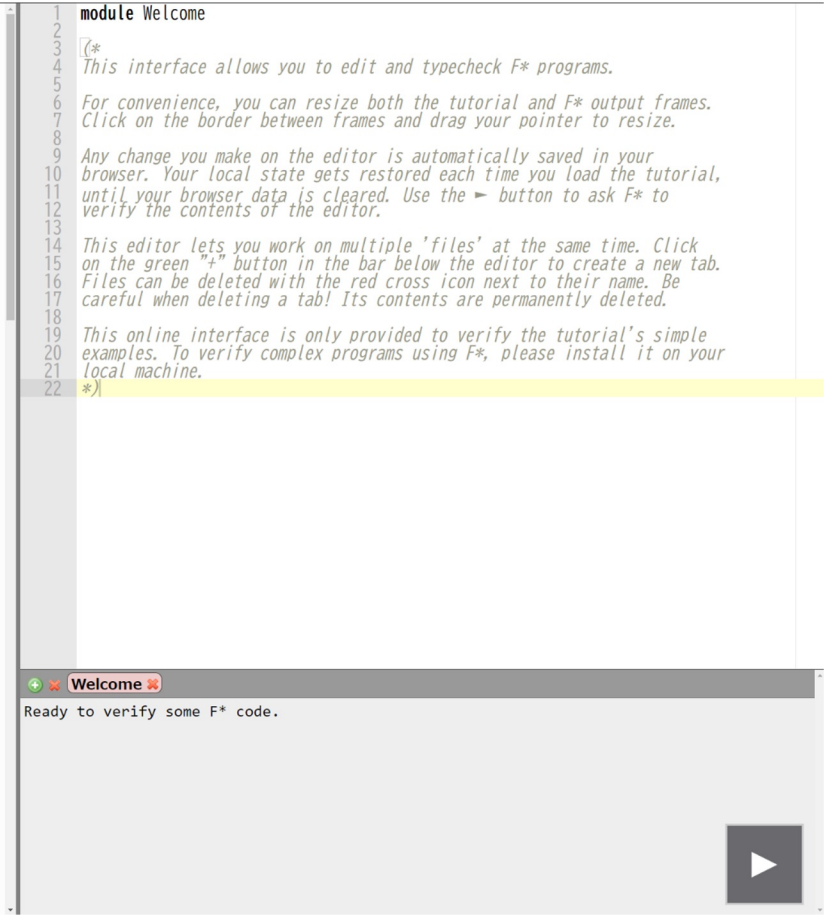
オンラインデモの使い方

https://fstar-lang.org/tutorial



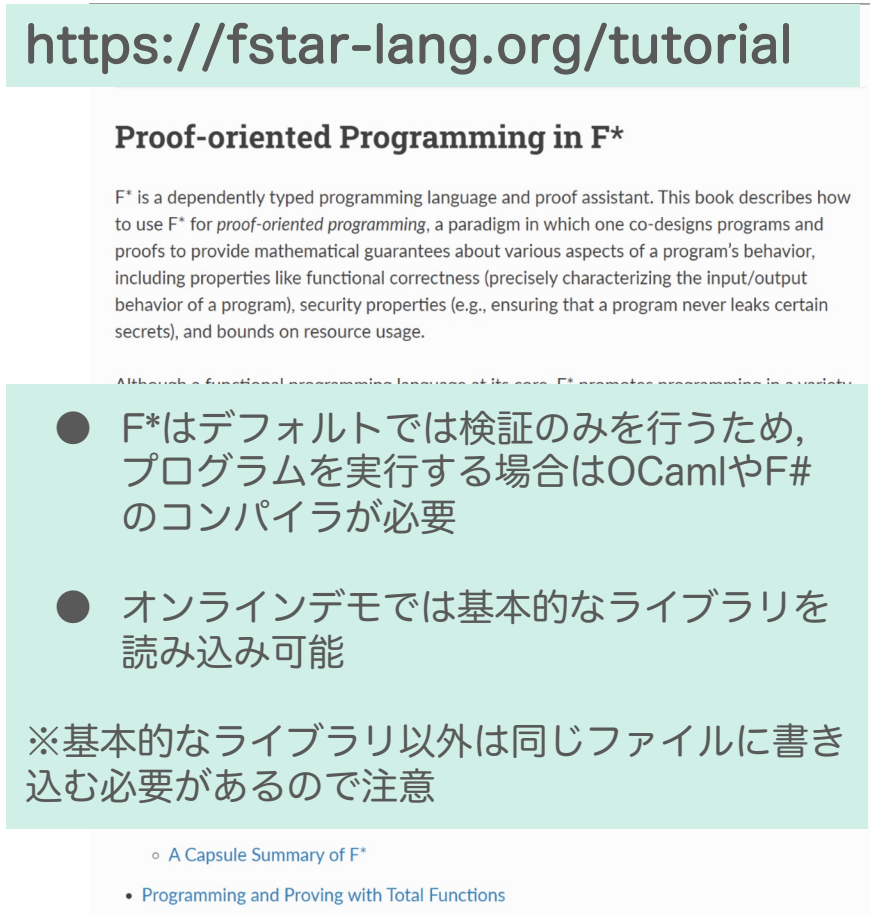
- F*はデフォルトでは検証のみを行うため，プログラムを実行する場合はOCamlやF#のコンパイラが必要
- オンラインデモでは基本的なライブラリを読み込み可能

※基本的なライブラリ以外は同じファイルに書き込む必要があるので注意



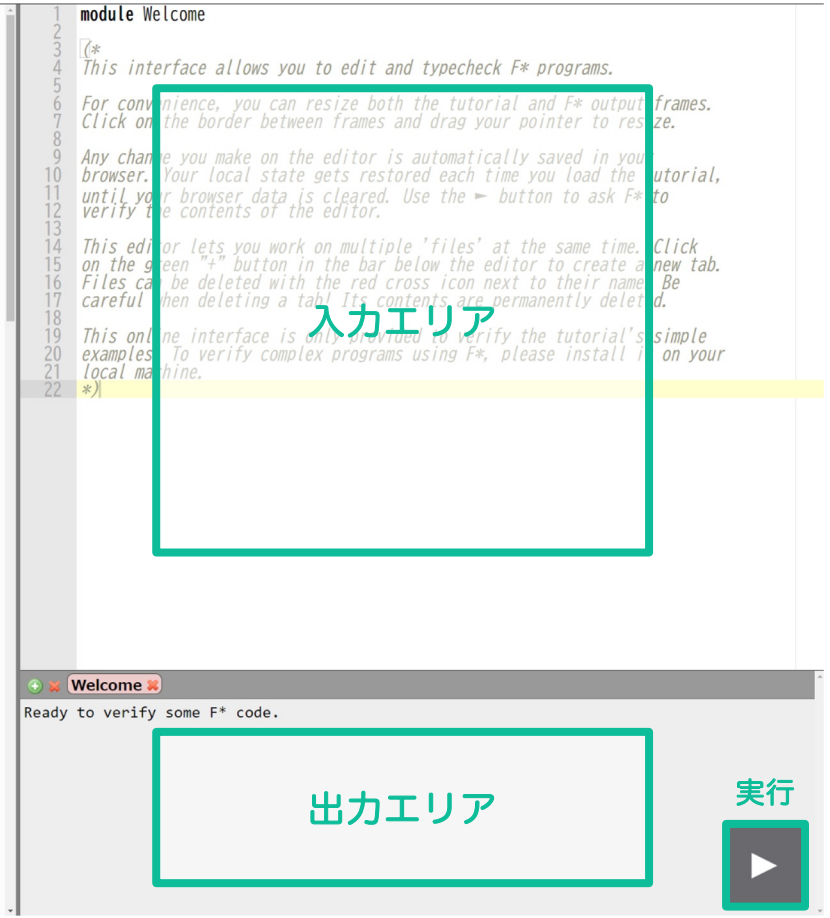
オンラインデモの使い方

https://fstar-lang.org/tutorial



- F*はデフォルトでは検証のみを行うため，プログラムを実行する場合はOCamlやF#のコンパイラが必要
- オンラインデモでは基本的なライブラリを読み込み可能

※基本的なライブラリ以外は同じファイルに書き込む必要があるので注意



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

- 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）ならばそのプログラムの中にエラーが存在しないと結論付けられる

プログラムが型安全 ⟺ プログラムがある種の振る舞いを起こさない

目次

基礎文法

F*のプログラムの基本構成

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

- プログラムはいくつかのモジュールから構成される
- モジュールは
  - （読み込むライブラリのリスト）
  - シグネチャ（定義に型を割り当てる）　`val f:t` など
  - 定義のリスト　`let f = e` や `type t = ...` など

  から構成される

基礎文法

- ブール演算

```
true
false
not    (* 否定 *)
&&     (* かつ *)
||     (* または *)
```

- 条件分岐

```
if condition then …
else …
```

- 関数の定義

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

(* x>yならtrueを返す関数 *)
let more_than (x:int) (y:int)
: bool
= x > y
```

int型に対して各種演算 -, +, /, %, <, <=, ... が定義されている

基礎文法

- match文による関数の定義

```
(* x=aならtrueをそれ以外ならfalseを返す関数 *)
let function (x:type) =
  match x with
    | a -> true
    | _ -> false
```

- 列挙による型の定義

```
(* 3種の要素からなる型 *)
type enumeration_type =
  | One : enumeration_type
  | Two : enumeration_type
  | Three : enumeration_type
```

- リファインメント型による定義

型の要素に対して成立する述語が付与された型

```
(* 自然数の型nat *)
let nat = x:int{x >= 0}

(* canRead fがtrueとなるようなfilename型の値を受け取りstring型の値を返す関数 *)
assume val read: f:filename{canRead f} -> string
```

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

剤余の演算子%が使えます

- 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
  それ以外はfalseを返す関数

演習2.1　解説

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

```
let even = x:int{x % 2 = 0}
```

- 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
  それ以外はfalseを返す関数

演習2.1　解説

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

```
let even = x:int{x % 2 = 0}
```

- 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

```
let addfunc (x:int{x >= 0}) (y:int{y >= 0}) : int = x + y
```

- （if文を用いて）int型の値を受け取り，0未満の値ならtrueを
  それ以外はfalseを返す関数

演習2.1　解説

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

```
let even = x:int{x % 2 = 0}
```

- 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

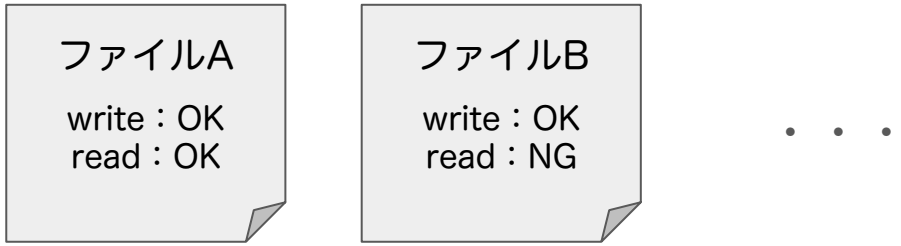
```
let addfunc (x:int{x >= 0}) (y:int{y >= 0}) : int = x + y
```

- （if文を用いて）int型の値を受け取り，0未満の値ならtrueをそれ以外はfalseを返す関数

```
let is_neg (x:int) : bool
= if x < 0 then true
  else false
```

最初の例：アクセスコントロールのモデル

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する



手順

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

最初の例：アクセスコントロールのモデル

1) アクセスポリシーを定義する

```
1  module ACLs
2    type filename = string
3
4    let canWrite (f:filename) =
5      match f with
6      | "demo/tempfile" -> true
7      | _ -> false
8
9    let canRead (f:filename) =
10     canWrite f
11     || f="demo/README"
```

コード1: アクセスポリシーのリスト（ACLs.fst）

ファイル名の型filenameをstring型で定義

ファイル名が"demo/tempfile"であればtrue
それ以外であればfalseを返す関数canWriteを定義

canWriteまたはファイル名が"demo/README"であればtrueそれ以外であればfalseを返す関数canReadを定義

demo/tempfile	demo/README
write：OK read：OK	write：NG read：OK

最初の例：アクセスコントロールのモデル

2) ファイルアクセスプリミティブを定義する

```
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
```

コード2: ファイルアクセスプリミティブ（FileIO.fst）

先ほど定義したアクセスポリシーモジュールを読込

canRead fがtrueと評価される（＝読み出し可能な）ファイル名fを引数に取り，string型の値（読み出した値）を返す関数readを定義

canWrite fがtrueと評価される（＝書き込み可能な）ファイル名fとstring型の値（書き込む値）を引数に取り，unit型の値を返す関数writeを定義

最初の例：アクセスコントロールのモデル

3) 安全性を検証する

```
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!"
```

"demo/tempfile"から読み出した値をv1に
"demo/README"から読み出した値をv2に格納

"demo/tempfile"に"hello!"を書き込み

コード3: クライアントコードの例（ClientCode.fst）

プログラムが型安全 ⟺ プログラムがアクセスポリシーを満たす

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

演習2.2　解説

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

```
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ます

このプログラムでは読み出し可能なファイル（demo/tempfile，demo/README）への読み出しと書き込み可能なファイル（demo/tempfile）への書き込みしか行っていないため，型検査が成功します

型検査が成功することを確認することで，プログラムがアクセスポリシーを満たしていることがわかります

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

演習2.3　解説

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

```
1  module ClientCodeNG
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     let v3 = read passwd in
11     write tmp "hello!"
```

コード4: 演習2.3の参考解答（ClientCodeNG.fst）

```
* Error 19 at ClientCodeNG.fst(10,18-10,24):
  - Subtyping check failed
  - Expected type f: ACLs.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(4,15-4,24)

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

canRead fを満たさないファイルをread関数に引数として渡しているので型エラーが出力される

演習2.3　解説

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

```
1  module ClientCodeNG
2    open FileIO
3    let passwd = "demo/password"
4    let readme = "demo/README"
5    let tmp = "demo/tempfile"
6
7
8
9
10
11
```

```
* Error 19 at ClientCodeNG.fst(10,18-10,24):
  - Subtyping check failed
  - Expected type f: ACLs.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(4,15-4,24)

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

に引数

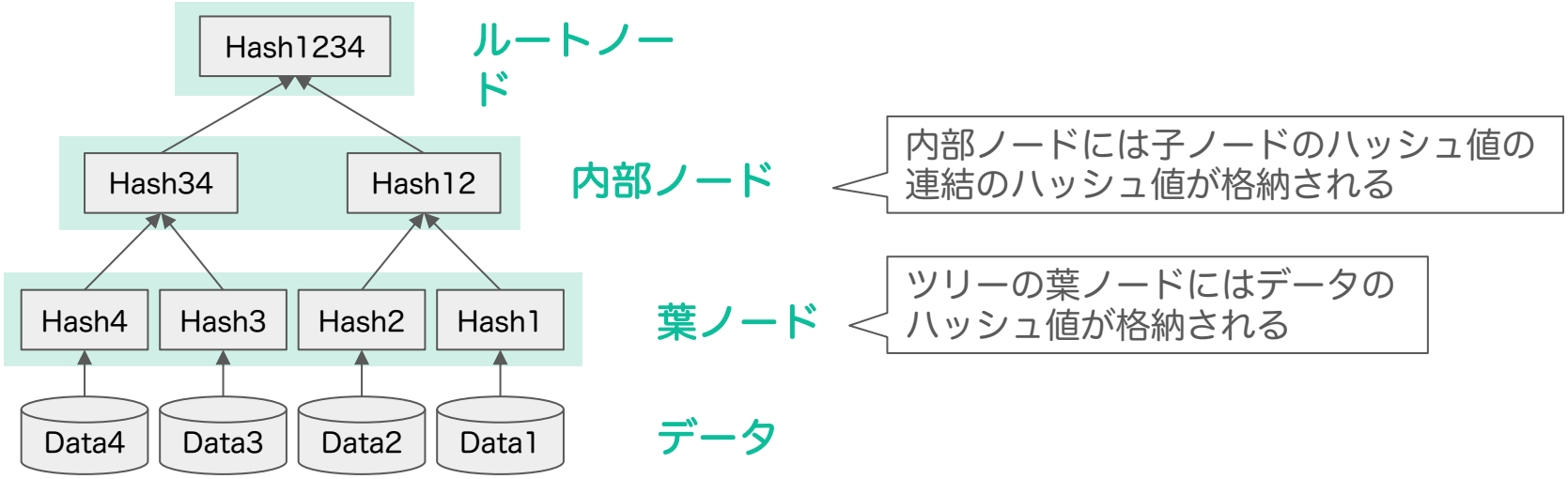
- 通常のプログラミングでは（ソースコードを注意深く見ない限り）プログラムを実行してテストしてから出ないとエラーに気づけない
- 型システムを用いることで実行前にエラーを発見できる！

目次

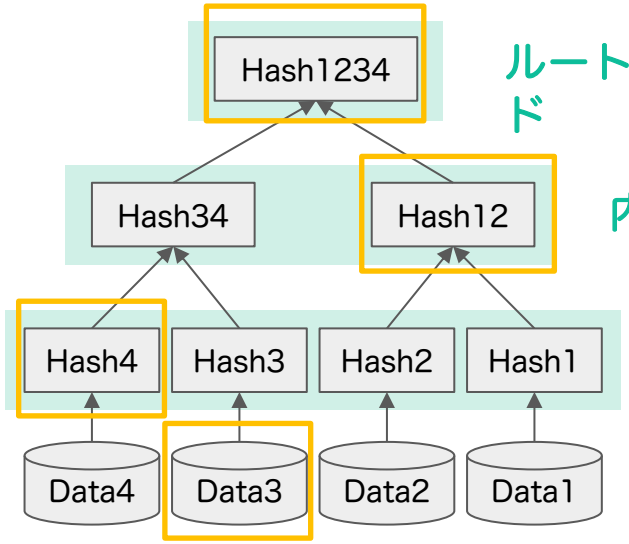
マークルツリーとは

Ralph Merkleによって導入されたデータ構造

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，PKIやブロックチェーンに応用されている



マークルツリーの利用例



ルートノード

内部ノード

内部ノードには子ノードのハッシュ値の連結のハッシュ値が格納される

葉ノード

ツリーの葉ノードにはデータのハッシュ値が格納される

データ

ブロックチェーン： Data ＝ トランザクション
証明書透明性：Data ＝ 証明書の発行履歴

マークルツリーのモデルとその検証

マークルツリーと周辺の機能をモデリングし，その安全性を検証する

手順

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) 証拠付きの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

マークルツリーのモデルとその検証

1) マークルツリーを定義する

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

長さnの文字列の型lstring nを定義

文字列を連結させる関数concatを定義

- ハッシュ関数の出力サイズ hash_size
- ハッシュ関数の出力の型 hash_t
- ハッシュ関数 hash
- ツリーに格納するデータの型 mtdata

を定義

コード5: マークルツリーの定義（MerkleTree.fst）1/2

マークルツリーのモデルとその検証

1) マークルツリーを定義する

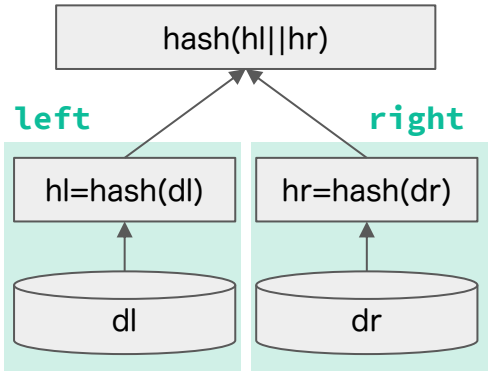
```
10
11  type mtree: nat -> hash_t -> Type =
12  | L:
13      data:mtdata ->
14      mtree 0 (hash data)
15  | N:
16      #n:nat ->
17      #hl:hash_t ->
18      #hr:hash_t ->
19      left:mtree n hl ->
20      right:mtree n hr ->
21      mtree (n + 1) (hash (concat hl hr))
```

データのハッシュ値が格納されている高さ0のマークルツリーとして葉ノードLを定義

左右のツリーから内部ノードNを定義

格納されている値は左右のツリーのルートノードの連結のハッシュ値

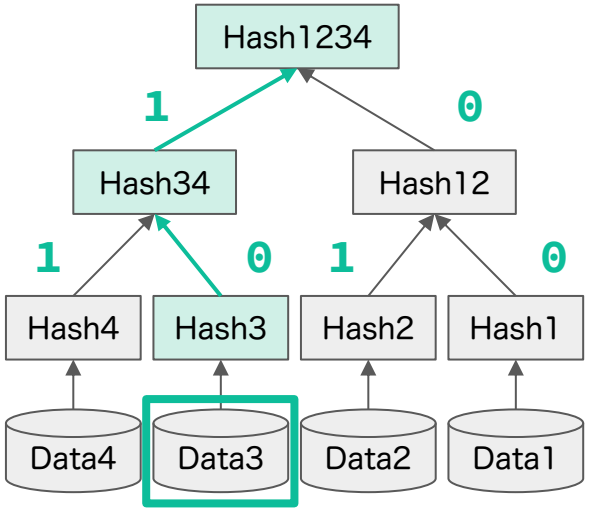
コード5: マークルツリーの定義（MerkleTree.fst）2/2



`mtree n h` はルートノードに格納されているハッシュ値がhの高さnのマークルツリーの型

マークルツリーのモデルとその検証

2) ツリーの各データへのアクセス関数を定義する



各データはルートノードからデータのハッシュ値を格納する葉ノードまでのパス（ブール値の列）を指定することでアクセス可能

このパスをデータIDと呼ぶ

10　（実際には `10` ではなく `true false`）

マークルツリーのモデルとその検証

2) ツリーの各データへのアクセス関数を定義する

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

コード6: データアクセス関数の定義（MTAccess.fst）

ブール値のリストでデータIDの型data_idを定義

データIDとマークルツリーを受け取りIDに対応するデータを返す関数getを定義

データIDを1ビットずつ読み込み，
- 1ならば左側のサブツリーに
- 0ならば右側のサブツリーに

移動し，葉ノードに到達したらそこに格納されているデータを返す

補足：文字列上のmatch文

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る

```
 9    = match did with
10    | [] -> L?.data tree
11    | b::did' ->
12        if b then get did' (N?.left tree)
13        else      get did' (N?.right tree)
```

コード6: データアクセス関数の定義（MTAccess.fst）

ブール値のリストdidに対して，
- didが空だったら
  - L.data treeを返す
- 文字列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'）をしていたら
  - もし b=true ならば get did' (N?.left tree)を返す
  - そうでなければ get did' (N?.right tree)を返す

（再帰的な定義）

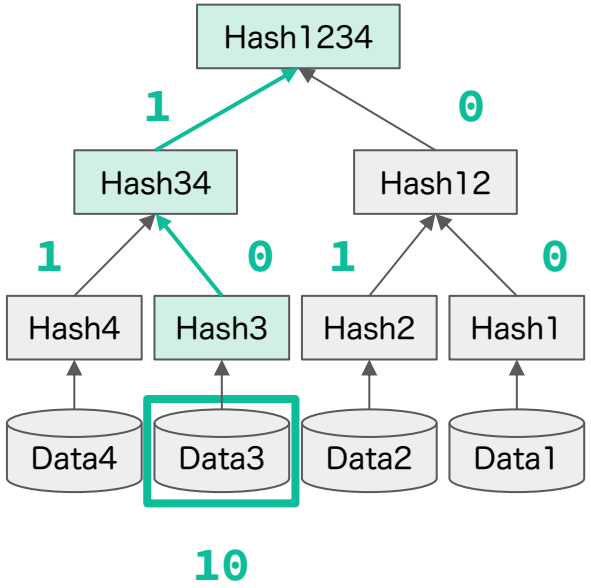
did=[true,false,true] のとき
did=b::did' として読み込むと
b=true,
did'=[false,true]

マークルツリーのモデルとその検証

2) ツリーの各データへのアクセス関数を定義する

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

コード6: データアクセス関数の定義（MTAccess.fst）



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

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

コード6: データアクセス関数の定義（MTAccess.fst）

演習3.1　解説

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

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

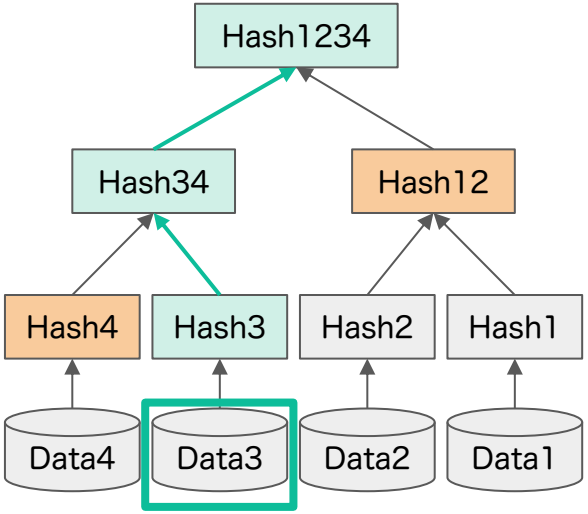
get h did tree

コード6: データアクセス関数の定義（MTAccess.fst）

マークルツリーのモデルとその検証

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

データと合わせて「特定のデータがツリー内に存在する」という証拠も出力するアクセス関数を定義



証拠はルートからデータまでのパスに沿った兄弟ノードのハッシュ値のリスト

Data3の場合は**[Hash12, Hash4]**

マークルツリーのモデルとその検証

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

データをデータIDとハッシュ値のリストでパッケージングした型data_with_evidenceを定義

```
1  module MTAccessEvidence
2    open MerkleTree
3    open MTAccess
4    module Lib = FStar.List.Tot
5
6    type data_with_evidence : nat -> Type =
7      | DATA:
8          data:mtdata ->
9          did:data_id ->
10         hashes:list hash_t {Lib.length did == Lib.length hashes} ->
11         data_with_evidence (Lib.length did)
```

p:data_with_evidenceに対し，
それぞれの要素は
- p.data
- p.did
- p.hashes

で指定可能

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

演習3.2

コード7の???部分を埋めて，証拠付きデータアクセス関数get_with_evidenceの型を定義してみましょう

```
12
13   let rec get_with_evidence (#h:_) ??? ???
14     :Tot ??? (decreases did)
15     = match did with
16     | [] ->
17         DATA (L?.data tree) [] []
18     | b::did' ->
19         let N #_ #hl #hr left right = tree in
20         if b then
21           let p = get_with_evidence did' left in
22           DATA p.data did (hr :: p.hashes)
23         else
24           let p = get_with_evidence did' right in
25           DATA p.data did (hl :: p.hashes)
```

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

演習3.2　解説

コード7の???部分を埋めて，証拠付きデータアクセス関数get_with_evidenceの型を定義してみましょう

```
12
13    let rec get_with_evidence (#h:_) (did:data_id)
                                (tree:mtree (Lib.length did) h)
14      :Tot (data_with_evidence (Lib.length did)) (decreases did)
15      = match did with
16      | [] ->
17          DATA (L?.data tree) [] []
18      | b::did' ->
19          let N #_ #hl #hr left right = tree in
20          if b then
21            let p = get_with_evidence did' left in
22            DATA p.data did (hr :: p.hashes)
23          else
24            let p = get_with_evidence did' right in
25            DATA p.data did (hl :: p.hashes)
```

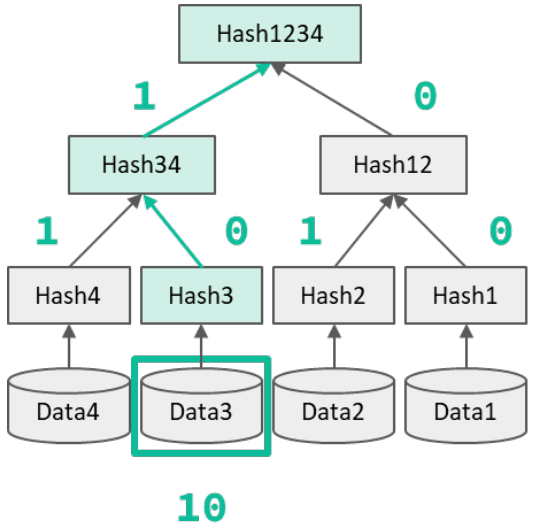
引数としてツリーのルートハッシュ，データID，ツリーを取り，証拠付きデータを返す関数

演習3.2　解説

コード7の???部分を埋めて，証拠付きデータ
get_with_evidenceの型を定義してみまし

```
12
13    let rec get_with_evidence (#h:_) (did:data_
                                (tree:mtree (Lib.
14      :Tot (data_with_evidence (Lib.length did
15      = match did with
16      | [] ->
17          DATA (L?.data tree) [] []
18      | b::did' ->
19          let N #_ #hl #hr left right = tree in
20          if b then
21            let p = get_with_evidence did' left in
22            DATA p.data did (hr :: p.hashes)
23          else
24            let p = get_with_evidence did' right in
25            DATA p.data did (hl :: p.hashes)
```

ルートノードから下に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



マークルツリーのモデルとその検証

4) 証拠を検証する関数を定義する

証拠からルートハッシュを計算する関数を定義

```
1   let tail #n (p:data_with_evidence n{n>0})
2     : data_with_evidence (n-1)
3     = DATA p.data (Lib.tail p.did) (Lib.tail p.hashes)
4
5   let rec compute_root_hash (#n:nat) (p:data_with_evidence n)
6     : hash_t
7     = let DATA d did hashes = p in
8     match did with
9     | [] -> hash p.data
10    | bit::did' ->
11        let h' = compute_root_hash (tail p) in
12        if bit then hash (concat h' (Lib.hd hashes))
13        else hash (concat (Lib.hd hashes) h')
```

長さnの証拠付きデータからそれぞれの要素の後ろ部分を抽出した長さn-1の証拠付きデータを返す関数

はじめにデータのハッシュ値を計算し，それ以降はデータIDの後ろ方向からハッシュを再計算

データIDに基づき，左側または右側の兄弟ハッシュを連結してそのハッシュ値を計算

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

演習3.3

ラスト演習です

コード8の???部分を埋めて，先ほど定義した関数compute_root_hashを用いて証拠が正しいかどうかを返す関数verifyを定義しましょう

```
14
15  let verify #h #n (p:data_with_evidence n) (tree:mtree n h)
16    : bool
17    = ???
```

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

証拠付きデータとマークルツリーを受け取り，証拠から計算されたハッシュ値とマークルツリーのルートハッシュが一致すれば`true`を，そうでなければ`false`を返す関数

値の一致は`=`で表現できる

演習3.3　解説

コード8の???部分を埋めて，先ほど定義した関数compute_root_hashを用いて証拠が正しいかどうかを返す関数verifyを定義しましょう

```
14
15  let verify #h #n (p:data_with_evidence n) (tree:mtree n h)
16    : bool
17    = compute_root_hash p = h
```

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

もちろんif文でも定義できます

マークルツリーのモデルとその検証

4) 証拠を検証する関数を定義する

F*では定理の形で性質を示すこともできる

```
18
19   let rec correctness (#h:hash_t) (did:data_id)
20                       (tree:mtree (Lib.length did) h)
21     : Lemma (ensures (verify (get_with_evidence did tree) tree))
22             (decreases did)
23     = match did with
24     | [] -> ()
25     | b::did' ->
26         let N left right = tree in
27         if b then
28             correctness did' left
29         else
30             correctness did' right
```

get_with_evidence関数が返す証拠はいつでも正しい（verify関数でtrueと評価される）ことを証明

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

マークルツリーのモデルとその検証

5) 安全性を証明する

マークルツリーの安全性として次を考える：

「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」

そのために次を示す：

「マークルツリーの中に存在しないデータの証拠（不正な証拠）が検証によって受け入れられる場合，ハッシュ関数の衝突を構成できる」

方針：

ハッシュ値が同じ値になるような2つの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

マークルツリーのモデルとその検証

5) 安全性を証明する

ハッシュ値が同じ値になる2つの異なる文字列のペアの型を定義

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

コード9: 安全性の証明（ProofSecurity.fst）の一部

マークルツリーのモデルとその検証

5) 安全性を証明する

安全性を表現する関数を定義

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

verify関数でtrueと評価される不正な証拠からhash_collision型の値を返す関数を作れることを示すことで安全性を証明

```
Verified module: MerkleTree
Verified module: MTAccess
Verified module: MTAccessEvidence
Verified module: VerifyEvidence
Verified module: ProofSecurity
All verification conditions discharged successfully
```

コード9: 安全性の証明（ProofSecurity.fst）の一部

目次

おわりに

本日は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を紹介しました

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取っ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もらえればと思います

演習中用スライド

ただいま演習時間です

そんな間にF*のセキュリティ応用について紹介（1/13）

F*の暗号実装用ライブラリも多数存在します

- **HACL***：C言語向けの暗号プリミティブライブラリ
- **ValeCrypt**：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**：HACL*とValeCryptを統合したフレームワーク

ただいま演習時間です

そんな間にF*のセキュリティ応用について紹介（2/13）

Implementing and Proving the TLS 1.3 Record Layer（S&P 2017）

Low*（C言語にコンパイル可能なF*のサブセット）を用いてTLS1.3を実装

ただいま演習時間です

そんな間にF*のセキュリティ応用について紹介（3/13）

HACL*: A Verified Modern Cryptographic Library（CCS 2017）

Low*で検証済みの暗号ライブラリを実装

ただいま演習時間です

そんな間にF*のセキュリティ応用について紹介（4/13）

Formally Verified Cryptographic Web Applications in WebAssembly
（S&P 2019）

HACL*を用いて Signal プロトコルを実装

ただいま演習時間です

そんな間にF*のセキュリティ応用について紹介（5/13）

EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider（S&P 2020）

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

ただいま演習時間です

そんな間にF*のセキュリティ応用について紹介（6/13）

HACL×N: Verified Generic SIMD Crypto (for all your favorite platforms) （CCS 2020）

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

ただいま演習時間です

そんな間にF*のセキュリティ応用について紹介（7/13）

A Security Model and Fully Verified Implementation for the IETF QUIC Record Layer（S&P 2021）

Low*を用いてQUICを実装

ただいま演習時間です

そんな間にF*のセキュリティ応用について紹介（8/13）

DICE*: A Formally Verified Implementation of DICE Measured Boot（USENIX Security 2021）

EverCryptを用いてMeasured bootプロトコルDICEを実装

ただいま演習時間です

そんな間にF*のセキュリティ応用について紹介（9/13）

DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code（Euro S&P 2021）

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

ただいま演習時間です

そんな間にF*のセキュリティ応用について紹介（10/13）

An In-Depth Symbolic Security Analysis of the ACME Standard（CCS 2021）

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

ただいま演習時間です

そんな間にF*のセキュリティ応用について紹介（11/13）

Noise*: A Library of Verified High-Performance Secure Channel Protocol Implementations（S&P 2022）

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

ただいま演習時間です

そんな間にF*のセキュリティ応用について紹介（12/13）

TreeSync: Authenticated Group Management for Messaging Layer Security（USENIX Security 2023）

DY*を用いてメッセージングレイヤセキュリティプロトコルを実装

ただいま演習時間です

そんな間にF*のセキュリティ応用について紹介（13/13）

Comparse: Provably Secure Formats for Cryptographic Protocols（CCS 2023）

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

ただいま演習時間です

そんな間にF*のセキュリティ応用について紹介（13/13）

HACL*: A Verified Modern Cryptographic Library（CCS 2017）

Low*で検証済みの暗号ライブラリを実装