特別研究報告書

暗号通貨向けストレージシステムにおける データ永続化処理の形式検証

指導教員：末永　幸平 准教授

京都大学工学部情報学科

伴野　良太郎

2021 年 2 月 2 日

暗号通貨向けストレージシステムにおける データ永続化処理の形式検証

伴野　良太郎

内容梗概

暗号通貨において、ブロックチェーンなどを実現するシステムにバグがあると、金銭的に大きな損害が発生しうる。バグによる損害が特に致命的である鉄道・航空機開発やクラウドシステムでは、形式検証を用いてその安全性が確保されてきた。同様にバグが致命的である暗号通貨でも形式検証は有益である。

本研究では、Tezos という暗号通貨で用いることが意図されている key-value ストレージシステム Plebeia のデータ永続化処理を、プログラム検証用プログラミング言語 F* を用いて検証した。この処理が正しく動作することで、Plebeia は、Tezos ブロックチェーンのデータを適切にディスクに保存し、また必要に応じて保存したデータを取得できる。データ永続化には、Plebeia の内部データである木構造をバイト列にエンコードしてディスクに書き込む関数群と、ディスクに書き込まれたバイト列を読み込みデコードして元の木構造に戻す関数群が必要である。本研究では、前者の関数群を用いて内部データ A をディスクに書き込んだ後、そのディスクから後者の関数群を用いて読み込みを行い B というデータが得られたときに、A と B が等価であることが成り立つための十分条件を明らかにし、実装の正しさを検証した。

F* は、OCaml とよく似た文法を持つ関数型プログラミング言語である。F* はカリーハワード同型対応を下敷きとしており、証明したい性質を型として表現し、その型を持つ純粋な全域関数として証明を書くと、F* に搭載された型検査器によってその証明が正しいことを自動的に検証できる。また F* にはコード抽出機能があり、これを用いることで F* コードを OCaml コードに変換できる。本研究ではまず、Plebeia において OCaml で実装されたデータ永続化処理を F* に移植した。次いで F* 上で証明を行い、その後コード抽出を行うことで、形式的に検証されたデータ永続化処理の OCaml 実装を得ることができた。

F* の証明内で用いる全ての関数は純粋かつ全域でなければならないが、データ永続化に関わる多くの関数はこれらを満たさないという問題があった。その理由は大きく二つに分けられる: (1) 関数がディスクへのアクセスを伴うため、純粋ではない。(2) Plebeia にはディスクの遅延読み込み機能があり、木

構造を再帰的にたどる最中に葉ノードが内部ノードに変わる場合がある。それゆえ次に訪れるべきノードが際限なく増加する可能性があり、再帰関数の停止性を形式的に判定できないため、関数の全域性を確認できない。(1) と (2) の問題を解決するために、本研究では各々次のような手法を使用した: (1) ディスクへのアクセスをヒープに対する読み書きとして表現し、このヒープを関数の引数と戻り値に明示的に含めた純粋な関数を別途定義した。その上で、新しく定義した関数と元々の関数が等価であることを証明し、以降の証明では新しく定義した関数を用いた。(2) ディスクの遅延読み込みを含むような木を再帰によって処理する場合は、自然数 `fuel` を引数に導入し、再帰呼び出しの度にこの値を 1 ずつ減らすことで再帰関数が止まることを保証した。またこれらの再帰関数の事前条件に、与えられた `fuel` で木の全てを読み込むことができるという仮定を含めた。

Plebeia ではディスクを 1 つの配列とみなし、そこから取り出した適切なサイズの部分配列に対してデータの書き込み・読み込みを行う。そのため F* においてこの配列を適切にモデル化し、部分配列への書き込みが既に書き込んだデータを上書きしないことや、書き込んだデータを読み込めば同じデータが得られることを示さねばならない。本研究では、F* の標準ライブラリにある `LowStar.Buffer` (`LSB`) モジュールを用いてこれを行った。`LSB` は C 言語における配列をモデル化したものとなっており、各要素への読み込み・書き込みを追跡しやすいように設計されている。この性質を用いて検証を容易に進めることができた。

書き込んだ木構造 A と読み出した木構造 B が等価であるかは、遅延読み込みを展開しつつ A と B を根から再帰的にたどり比較することによって調べられる。本研究ではこれを行う関数を `equivalent_nodes` という名前で定義し、これを用いて証明を行った。一方で Plebeia には等価性判定のための `equal` という関数が定義されているが、その名前に反して、この関数は引数の完全な等価性を調べていない。そこで本研究では、`equivalent_nodes` と `equal` の A と B に関する等価性判定が一致する十分条件を求めた。

検証は 17,036 行の F* コードからなり、その型検査の実行には 3,462 秒を要した。コード抽出を行って得られた OCaml コードは、元々 Plebeia に付属していた全てのテストを元々の実装と同程度の時間で通過した。

Formal Verification for Data Persistence Process of a Storage System for Cryptocurrency

Ryotaro Banno

Abstract

Bugs in blockchain systems for cryptocurrencies can cause severe economic damage. In the development of railways, airplanes, and cloud systems, since their bugs may lead to an irreparable loss, *formal verification* has been used to ensure their safety. It is also beneficial for cryptocurrencies as well, since their safety is also crucial.

In this thesis, we formally verify the process of data persistence in Plebeia by using F*. Plebeia is a key–value storage system planned to be used in cryptocurrency Tezos, and F* is a programming language aimed at program verification. By formally verifying the data-persistence functionality of Plebeia, we can guarantee that it stores data for Tezos blockchain to disks and also retrieves the data from them correctly. The data-persistence functionality is implemented by two functions: one is for encoding internal data of Plebeia, which have a tree-shaped structure called *Plebeia trees*, into a byte sequence, and writing it to disk. The other is for reading a byte sequence from disk and decoding it into a Plebeia tree. We verified the following property: if the former function writes a Plebeia tree A and then the latter reads the stored data to a Plebeia tree B, A is equal to B.

F* is a functional programming language, whose syntax is very similar to that of OCaml. Based on the Curry–Howard isomorphism, we can automatically prove a statement using its type checker by expressing it as a type and writing its proof as a pure and total function of the type. F* also allows us to extract OCaml code from F* code. In this thesis, we first port an existing OCaml implementation of the data-persistence functionalities of Plebeia into F*. Then, we proved its correctness using F* and extracted OCaml code to obtain its formally verified implementation.

Although all the functions used in a proof in F* must be pure and total, many functions related to data persistence do not satisfy this condition due to the following two reasons: (1) They have to access disks, which makes them impure; (2) Since Plebeia lazily reads the disks, it converts leaf nodes into internal ones

while scanning a tree recursively; therefore, the number of the nodes to visit may increase indefinitely, and we cannot check whether they terminate, which makes them not total.

To solve these problems (1) and (2), we used the following strategies. For the issue (1): We modeled accesses to disk as accesses to heap by a pure function that explicitly takes a store as an argument and returns the updated store. Then, we proved that the original function and the new one are equivalent, and we used the latter in other proofs. For the issue (2): To the function that scans a tree a part of which is read lazily from disk, we added a natural number `fuel` as its argument; the value of `fuel` is subtracted by 1 at every recursive call to ensure its termination. We included, to its precondition, an assumption that given `fuel` is enough to fully load the tree.

Plebeia regards a disk as an array, and writes to and reads from a subarray of proper size. Therefore, we need to model the array in F* and prove (1) that a write to the subarray does not overwrite its already written data and (2) that read data are the same as the previously written one. In this thesis, we used `LowStar.Buffer` (`LSB`) module in F* standard library. `LSB` is designed to be a model of arrays in C and to track element-wise reading and writing easily, which makes our verification simple.

We can check if the written data A and the read data B are equal by comparing their nodes recursively from their roots, performing lazy loading. We defined a new function `equivalent_nodes` for this check and proved their equality using it. While Plebeia defines another function `equal` for equality check, it does not fully check the equality despite its name. In this thesis, we found a reasonable sufficient condition such that `equivalent_nodes` and `equal` provide the same results if A and B are applied to them.

Our verification consists of 17,036 lines of F* code, and it takes 3,462 seconds to type check our F* code. The extracted OCaml code passes all the tests originally included in Plebeia as fast as the original implementation does.

暗号通貨向けストレージシステムにおける データ永続化処理の形式検証

目次

1 序論

互いの信用を必要としない分散型送金プラットフォームとして、Bitcoin [1] や Ethereum [2] といった、ブロックチェーンを用いた暗号通貨が大きな成功を収めている。暗号通貨の取引において使用されるソフトウェアに脆弱性があると、その脆弱性をついた攻撃が行われ、金銭的な損害を生じる恐れがある。過去には実際にそのような攻撃が行われ、多額の損失を出した [3]。コード中のバグを防ぐ手法としてはソフトウェアテストが挙げられるが、全ての入力や内部状態についてテストすることは現実的に不可能であるため、バグを見逃してしまう危険性がある。

このようなバグによる人的・金銭的なダメージが致命的になりうる、いわゆるミッションクリティカルなソフトウェアでは、バグの混入を防ぐために形式検証が用いられてきた [4,5]。同様に暗号通貨でも形式検証による安全性の保証が有益であると考えられ、ブロックチェーンの合意形成やスマートコントラクトに関する形式検証が研究されてきた [6,7]。一方で暗号通貨システムの実装そのものの安全性については研究が少ないが、これもまた重要なセキュリティ上の問題である。

Plebeia [8,9] は Tezos [10] という暗号通貨向けに OCaml で開発された新しいストレージシステムである。現在 Tezos では内部データを保存するために Irmin2 というストレージシステムが採用されている。Plebeia は Merkle Patricia Tree の変種（Plebeia tree）を用いることで Irmin2 よりも効率的に情報を保存できるという特長を持つ。ストレージシステムは、内部に保持するデータを永続化するために、適宜内部データをディスクに書き込まねばならない。Plebeia では Plebeia tree を適切にエンコードしてディスクに書き込み、また適宜これを読み込んでデコードし tree を再構築する必要がある。

本研究では、Plebeia のデータ永続化処理について形式的に検証を行い、そのエンコードとデコード処理の実装が正しいことを示した。より具体的には、ある Plebeia tree A をディスクに書き込んだ後、そのディスクから読み込みを行って Plebeia tree B が得られたときに、この A と B が等価であることが成り立つための十分条件を明らかにした（図 1）。

この証明を行うために、我々は F* [11] を用いた。F* は OCaml とよく似た文法を持つ関数型プログラミング言語であり、プログラム検証を行うための機

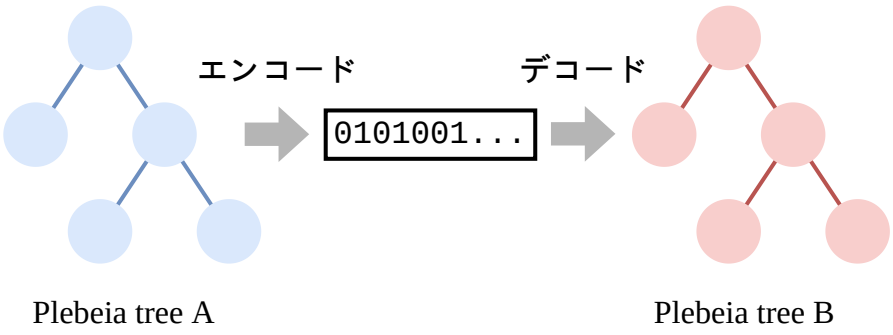

図 1　Plebeia tree A をエンコードしてディスクに書き込み、それを読み込んでデコードして Plebeia tree B を得る。本研究では、適切な条件のもとで A と B が等価であることを示した。

能を多数搭載している。F* はカリーハワード同型対応を下敷きとしており、証明したい性質を型として表現し、その型を持つ純粋な全域関数として証明を書くと、F* に搭載された型検査器によってその証明が正しいことを自動的に検証できる。検証された F* コードは、F* コンパイラの機能によって OCaml コードへと変換することができる。

我々はまず、OCaml で書かれている Plebeia のデータ永続化処理を F* に移植し、その後に F* プログラムとして証明を記述した。この証明の正しさを F* の型検査器によって自動的に検証した後、OCaml コードへと変換した。変換されたコードは、検証されていない他の OCaml コードとリンクし動かすことができる。我々はリンクして得られた実行可能バイナリが、元々の Plebeia にある全てのテストを通過することを確認した。

本報告書は次のような構成になっている。第 2 章で F* の概要を述べ、第 3 章で Plebeia について説明する。第 4 章で F* による Plebeia の検証について述べる。その後第 5 章で検証の実行について述べる。第 6 章で関連研究について議論し、第 7 章で結論と今後の課題を述べる。

2　F*

F* [11] はプログラム検証のための関数型プログラミング言語である。この章では F* の機能について、特に我々が検証に用いたものに焦点を当てて概説する。

2.1 文法

F* の文法は OCaml と非常によく似ている。たとえば、整数のリストを引数に受け取りその総和を計算して返す関数sumは、F* では次のように定義できる：

```
(* 関数の型宣言: 整数のリストを受け取り、整数を返す *)
val sum : list int -> int
(* 関数の定義 *)
let rec sum = function
 | [] -> 0 (* 入力が空の場合は 0 を返す *)
 | x :: xs -> sum xs + x
     (* そうでなければ、残りの合計に先頭を足す *)
```

多相型 t の型パラメータは、OCaml では `('a, 'b, ...) t` と書くのに対して、F* では `t 'a 'b ...` と書く点で異なる。

2.2 エフェクト

F* における関数型には、次のようにエフェクトを付記できる。

```
引数の型 -> [エフェクト] 戻り値の型
```

エフェクトはその関数が呼び出されたときに起こりうる副作用を表しており、その関数内で使用できる関数に制約を設ける。F* には次のようなエフェクトが定義されているほか、プログラマが定義することもできる：

PURE・Tot　関数がいかなる副作用も持たず、またいかなる引数についても必ず停止することを表す。**Tot**は**PURE**から派生したエフェクトで、関数を呼び出す際の事前条件と事後条件に制約を設けない。エフェクトを明記しない場合、デフォルトでは**Tot**が使用される。

GHOST　関数がいかなる副作用も持たず、またいかなる引数についても必ず停止することを表す。**PURE**や**Tot**と異なり、**GHOST**エフェクトやそれから派生したエフェクトを持つ関数は F* 上でのみ有効な関数であって、コード抽出時に抽出されない。

ST・St　関数が、参照などを通じてメモリへの書き込みや読み込みを行いうることを表す。**St**は**ST**から派生したエフェクトで、関数を呼び

出す際の事前条件と事後条件に制約を設けない。

他に 2.3 節で説明するLemmaエフェクトなどもある。

使用したエフェクトに応じて、F* では型宣言を詳細化できる。たとえばSTエフェクトの場合、関数の呼び出し前後でのメモリに関する条件について詳細化できる。

```
val f : r:ref int -> ST unit
  (* 事前条件：参照 r の値が正であること *)
  (requires (fun h -> sel h r > 0))
  (* 事後条件：参照 r の値が関数呼び出し前後で大きくなること *)
  (ensures (fun h _ h' -> sel h' r > sel h r))
let f r =
  r := !r + !r (* 参照 r の値を 2 倍にする *)
```

2.3 補題の定義とその証明

F* において補題を定義・証明する際にはLemmaエフェクトを用いる。例えば正数 n, m を足し合わせると正数が得られるという補題は次のように表現できる。

```
val lem : n:int -> m:int -> Lemma
  (requires (n > 0 ∧ m > 0)) (* 事前条件: n と m が正である *)
  (ensures (n + m > 0))      (* 事後条件: n + m が正である *)
```

補題に関する証明は、unit 型を持つ関数として表現する。F* は、Z3 という SMT ソルバを用いて検証条件を解決することによって、その証明が正しい否かを自動的に検証する。ここでは補題が単純なため、()とすればよい。

```
let lem n m = ()
```

なおLemmaエフェクトはGHOSTエフェクトから派生しているため、補題の定義や証明において使用する全ての関数は、PUREまたはGHOSTエフェクト、あるいはそれから派生したエフェクトである必要がある。

2.4 再帰関数の停止性の証明

Totエフェクトを持つ関数を定義する場合、その関数は必ず停止しなければならない。しかし一般に、ある関数が停止するかを判定する問題は決定不能である。F* の場合、問題になるのは再帰関数の停止性を証明しようとするときである。F* はその停止性を判定するために、次のような戦略を用いる。

- まず F* は、全ての再帰関数呼び出しにおいて、引数が構造的に小さくなっているかを証明しようとする。例えば先程のsum関数では、再帰呼び出しにおける第一引数のリストが元のリストよりも短くなるため、F* は自動的に停止性を証明できる。
- プログラマは、どの引数が構造的に小さくなっているのか F* に教えることができる。この場合、F* は教えられた引数が本当に小さくなっているのか確かめようとする。例えば、関数 sum に引数 c をアキュムレータとして追加して、次に掲げるような関数 sum' を作る。この関数の第一引数は減少していないため、停止性を自動的に証明することはできない。

```
val sum' : int -> list int -> Tot int
let rec sum' c = function
  | [] -> c
  | x :: xs -> sum' (x + c) xs
```

しかしながら、次のように型宣言に(decreases v)を付け加えて第二引数が減少していることを表明すれば、F* はその停止性を証明できる。

```
val sum' : int -> v:list int -> Tot int (decreases v)
```

2.5 F* ソースファイルとインタフェースファイル

F* には二つのファイル形式がある。1 つ目はソースファイルであり、拡張子.fst を持つ。2 つ目はインタフェースファイルであり、拡張子.fsti を持つ。ソースファイルには関数の宣言・定義や補題の定義、及びその証明などが書かれている。インタフェースファイルには、F* ソースファイルには含まれない関数の型宣言などが含まれている。インタフェースファイル中の型宣言は、証明中では仮定として扱われる。

2.6 コード抽出

F* コンパイラの機能を用いて、F* ソースファイルとインタフェースファイルから、自動的に OCaml ソースファイルを抽出できる。抽出されたファイルには、F* 上で定義された関数のうち、GHOSTエフェクト（及びその派生）を持たない全ての関数と等価な OCaml の関数が定義される。例えば先述のsum関数を抽出すると、次のような OCaml コードになる（見やすさのためにインデント・改行を補っている）：

```
let rec (sum : Prims.int Prims.list -> Prims.int) =
  fun uu___ ->
    match uu___ with
    | [] -> Prims.int_zero
    | x::xs -> (sum xs) + x
```

3 Plebeia

3.1 概要

Plebeia は OCaml によって実装された Tezos 用の key–value ストレージシステムである。内部データの保持には、Tezos のブロックチェーンに適するように Merkle Patricia tree を特殊化したデータ構造（Plebeia tree）を用いており、現在 Tezos に使われているストレージシステムである Irmin2 よりも効率的にデータを保持できる [9]。Plebeia は GitLab 上で開発されている [8]。

我々は検証に際して、Plebeia のコミット `44532dd` を元に行った。最新のコミットへの追従は今後の課題である。

3.2 Plebeia tree

Plebeia tree は Plebeia の内部データを保持するために使用されているデータ構造である [12]。Plebeia tree は Merkle Patricia tree をベースに、Tezos ブロックチェーンに適合するように改良が加えられている。

OCaml を用いると、Plebeia tree は次のように定義できる。ただし簡単のため、実際の実装を一部簡略化している：

```
type side = L | R
type key = side list
type node =
  | Disk of index
  | View of view
and view =
  | Leaf of value * index option * hash option
  | Bud of node option * index option * hash option
  | Internal of node * node * index option * hash option
  | Extender of key * node * index option * hash option
```

Diskはノードの遅延読み込みを表す。引数のindexはディスクのどこから読み込むべきかを表す。詳しくは 3.3 節において説明する。

Viewは tree を実質的に構成している。Leafは葉ノードを表し、ストレージシステムが保持すべきデータを持っている。Bud・Internal・Extenderは内部ノードであり、子ノードをそれぞれ 1 つ・2 つ・1 つ持つ。ただしBudのみ子ノードを持たず葉ノードBud (None, _, _)になる場合がある。これらが持つindex optionとhash optionは、そのノードが既にディスクに書き込まれている場合に、そのディスク上の位置と計算されたハッシュ値を保持する。特にindex optionの値がSome iであるときに、そのノードを「iで indexed なノード」と呼ぶことにする。

例えば、図 2 のような Plebeia tree は次のようなコードにより構成できる。ただしkey・index option・hash optionは省略している。

```
Bud (Some (Internal (Leaf A, Extender (Leaf B))))
```

任意のノードnは、そのnを根とする Plebeia tree を示しているとみなせる。これを「nが指す木」と呼ぶことにする。

3.3 Node_storage モジュール

Node_storageモジュールは Plebeia のデータ永続化を担っており、Plebeia tree をバイナリ列に変換してディスクへと書き込むほか、逆にディスクからバイナリ列を読み込んで Plebeia tree を再構築する。Plebeia では

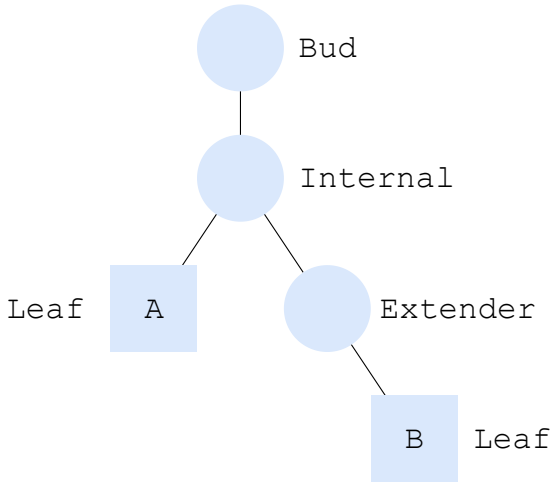

図 2　Bud (Some (Internal (Leaf A, Extender (Leaf B)))) が表す Plebeia tree（key・index option・hash option は省略）。

node_storage.ml ファイルにおいて実装されている。

```
let open Node_storage in
(* n0 をディスクに書き込む。書き込んだ先として i を得る *)
let n1, i, _ = commit_node context bud_cache n0 in
(* i からノードを読み込み n2 を得る *)
let n2 = View (load_node context i) in
(* n0 と n2 の等価性を確認する *)
equal context n0 n2 = Result.Ok ()
```

プログラム 1　ノード n0 をディスクに書き込み、書き込んだ先 i から読み込んで n2 を再構築し、n0 と n2 の等価性を確かめる OCaml コード

プログラム 1 は Node_storage 内で定義される関数を用いて、ノード n0 をディスクに書き込み、書き込んだ先から読み込みを行ってノード n2 を再構築し、n0 と n2 が等しいことを確かめる OCaml コードである。以下ではこのコードで使用されている関数を中心に、Node_storage の内容について簡単にまとめる。

Node_storage は Plebeia tree のバイト列へのエンコードとデコードを主に行い、実際のファイルへの書き込みは Storage モジュールと Xcstruct モジュールにある関数を呼び出すことによって行う（図 3）。

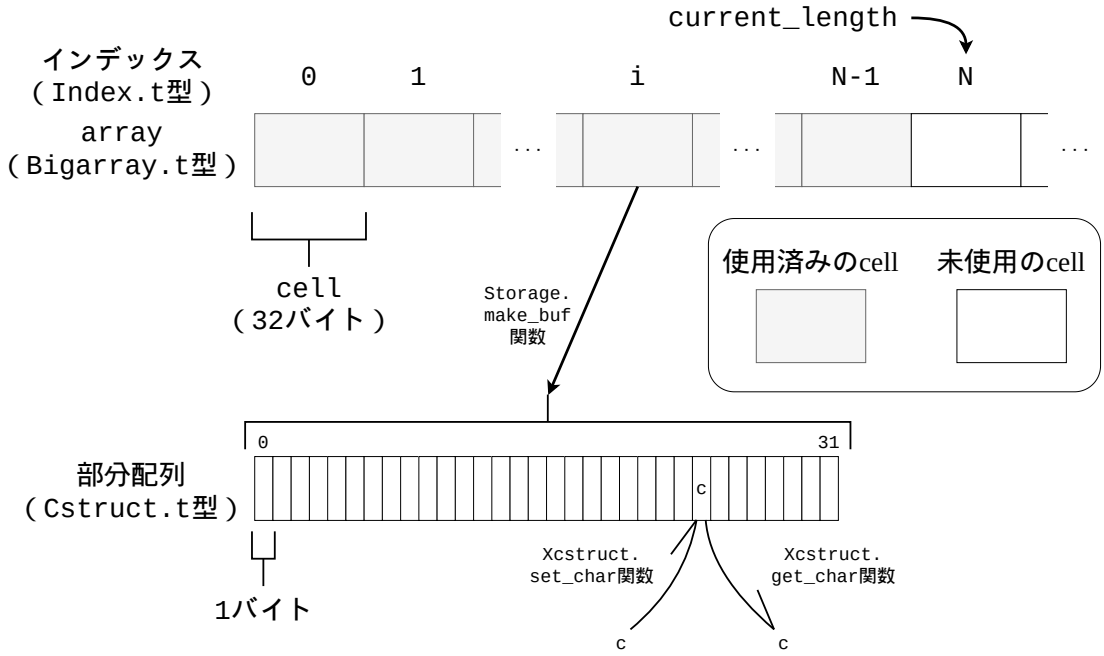

図 3　Node_storageモジュールが使用するStorageモジュールとXcstructモジュールの概要。

Storageモジュールは書き込むべきディスクへのアクセスをStorage.t型を介して提供する。Storage.t型はディスク本体を表す memory-mapped ファイルへの参照（array : Bigarray.t）と、現在そのファイルをどこまで使用しているかを表すポインタ（current_length : Index.t）を持つ。Storageモジュールはこのファイルを大きな一次元配列としてみなし、一つの要素が 32 バイトであるとする。この配列の各々の要素を Plebeia では cell と呼ぶ。任意のノードは、整数個の cell を使用して格納される。Index.t型は一つの cell を指し示すための添字（インデックス）を表す型で、符号なし 32bit 整数値で実装される。ただしその値の内 256 個は、特殊な cell に関するタグとして用いられる。したがってファイルのサイズは最大 $(2^{32} - 256) \times 32 = 137\,438\,945\,280$ バイト（約 128 GiB）になる。ユーザはStorage.new_index関数を用いて新しい cell を作る。この cell はストレージの末尾、すなわちcurrent_lengthが指し示す場所に作られる。この時current_lengthは 1 増加する。新しい cell は必ずストレージの末尾に作られ、一度作られた cell が削除されることはない。したがってcurrent_lengthは Plebeia の動作に従って単調に増加する。

翻ってXcstructモジュールは、arrayから切り出された部分配列に対して 1

バイトごとに書き込み・読み込みを行うためのインタフェースを提供する。Storage.make_buf関数を用いることで部分配列（Cstruct.t型）を得ることができる。これを例えばXcstruct.get_charに渡せば、その部分配列中の任意の1バイトを取得することができる。あるいはXcstruct.set_charに渡せば1バイトを書き込むことができる。

ノード書き込みのエントリポイントはcommit_node関数である。commit_nodeは以下の型を持つ：

```
Context.t -> Bud_cache.t -> node
  -> node * Index.t * Hash_prefix.t
```

第一・第二引数は今使用しているストレージやキャッシュの情報を持つ。これらは本研究ではそれほど重要ではないため、以下ではcontext・bud_cacheという名前で各々の型をもった変数を使用し、これらの変数は適切に初期化されているものとする。第三引数が、いま書き込むべきノードnを表す。戻り値のnodeはnと同じ構造を持ちながらも indexed なノードであり、Index.tはnを書き込んだ先の添字である。commit_nodeでは、nを再帰的にたどりながら、次のように書き込みを行う。

- nがDisk iの場合は、すでにnはiに書き込まれているため、新たに書き込む必要がない。終了する。
- nがView vの場合は、次の二つの場合に分ける。
  - nが indexed な場合、すでにディスクに書き込まれているため、新たに書き込む必要がない。終了する。
  - nがまだディスクに書き込まれていない場合、vの内容に従って書き込む。特にvがBud (Some _)・Internal _・Extender _の場合は子ノードを持つため、再帰的にcommit_nodeを呼び出す。

ノードの読み込みはload_node関数を主軸に行われる。この関数は読み込むべきインデックスを引数に受け取り、その引数でparse_cellという別の関数を呼び出す。parse_cellは引数のインデックスにあるバイト列をディスクから読み込み、Plebeia tree を再帰的に再構築することによってノードの読み込みを行う。ただし辿れる木全体を読み込むわけではなく、Internalに遭遇した場合は、子ノードをDiskとしてそれ以上は読み込まない。これは必要な部分だけを高速に読み込むことを可能にするため、パフォーマンス上の優位性がある。木

全体を読み込むためにはload_node_fullyやload_node_fully_for_testなどの別の関数を用いる。これらの関数は内部でload_nodeを再帰的に呼び出す。なお、今回の検証ではload_node_fully_for_testを用い、load_node_fullyについては検証を行わなかった。この点は今後の課題である。

ノードの等価性を比較するためにequal関数が定義されている。この関数は引数に受け取った二つのノードを再帰的にたどり、そのノードが同じ構造をしているか調べる。この関数はパフォーマンス上の優位性から末尾再帰関数として定義されている。そのため形式検証は複雑になる。また左右辺がともに遅延読み込み対象の場合は、その読み込み箇所が同じかどうかを調べるのみで、実際の読み込みは行わない。したがって、実際に読み込み処理を行った場合には等価であると分かる場合でも、equalがResult.Ok ()を返さない場合がある。

なお実際の実装では、Diskはインデックスの他に extender witness と呼ばれる要素を引数に持ち、そのインデックスから読み込みを行った際に得られるノードが extender であるか否かを表すが、本報告書では省略している。また元々の Plebeia の実装では、equal関数において左右辺がともにDiskの場合に、インデックスの等価性のみならず extender witness の等価性まで求めているが、検証中にこれは不要であることが分かったため、検証したコードではこの処理を削除した。この変更は Plebeia の元々の実装にも取り込まれる予定である。

4 F* による Node_storage モジュールの検証

4.1 概要

この章では、我々がどのようにNode_storageモジュールを F* において検証したかを述べる。

我々が実施した F* による検証は、次のような構造になっている。OCaml コードにおいてNode_storageモジュールを定義している node_storage.ml ファイルの内容は、F* ソースファイルとして node_storage_fst.fst ファイルに移植された。このファイルにはNode_storage_fstモジュールが定義されており、元々Node_storageモジュールに存在していた関数のうち、検証に使用するものが含まれている。また node_storage_fst.fst ファイル以外にもいくつかの F* ソースファイルが存在している。これらはNode_storage_fstにおける証明を行うために必要である。さらに、Node_storageモジュール

が依存しているその他のモジュールについて、そのインタフェースを F* インタフェースファイルとして宣言している。これには例えば segment.fsti や context.fsti などがある。以上のファイル群によって、Node_storageモジュールの検証ができる。

我々が行った検証は Sato [12] が導入した F* コードに基づいている。Sato は、node.ml と cursor.ml を F* を用いて検証し、Plebeia tree のノードについて満たされるべき不変条件がこれらのファイル中で保たれることを示した。Sato は node.ml と cursor.ml を各々 node.fst と cursor.fst に移植した。我々の検証でも Sato が実装したこれらの F* ソースファイルを利用し、Node_storageモジュール内でノードを新しく作る場合にはその不変条件が満たされるようにした。また外部からNode_storageモジュールに対して入力されるノードは、その不変条件を必ず満たしていると仮定した。

本検証で証明すべき事項は、直観的には、プログラム 1 が必ずtrueと評価されることと定式化できる。より正確な定式化は 4.5 節にて導入する。

Plebeia tree の等価性は、木について自然に再帰することによって調べることができる。これを行う関数として、我々はequivalent_nodes関数を導入する。一方で Plebeia が元々定義しているequal関数は、equivalent_nodesと異なる意味論を持つ。これらの対応関係については 4.4 節にて議論する。

Node_storageモジュール内の関数が呼び出す外部モジュールの関数の型や挙動は、F* インタフェースファイルに書くことで仮定しなければならない。証明のためには特に、ディスクとして扱う memory-mapped file に対するアクセスを提供するStorage及びXcstructモジュールに関する仮定が重要である。この点については 4.3 節にて議論する。

commit_node・load_node・equal関数はいずれも全域純粋関数ではないため、定理中で直接参照することができない。そこで、これらの関数と同じ結果を返すような純粋関数を F* 上で定義し、これを証明中では用いる。この関数の定義方法については 4.2 節にて論じる。

以上の内容を使って証明を完成させることができる。証明は付録（A.1）にて述べる。

F* コードを用いて検証を行った後は、これを OCaml コードに抽出し、Plebeia に組み込んで動作させる必要がある。これについては 4.6 節で述べる。

4.2　読み込み・書き込み関数の全域純粋関数への変換

commit_node・load_node・equal関数などについて証明を記述するためには、当然これらの関数を証明中で呼び出さねばならない。しかしこれらの関数はいずれもディスクの読み込み・書き込みを伴うため、STエフェクトを持つ関数である。そのような関数は証明中では使用できない。

そこで、STエフェクトを持つ各々の関数と同じ結果を返すような全域純粋関数を F* 上で定義し、証明中ではこれを用いる。この新しく定義する関数を、以下ではプレフィクス "model_" を用いて区別する。例えばequalの全域純粋関数バージョンはmodel_equalである。またcommit_nodeやparse_cellなど、内部に再帰関数を定義しているものについては、その再帰関数に関して全域純粋関数を定義する。したがって各々model_commit_node_aux・model_parse_cell_recとなる。

これらの関数を定義する際には、元々の関数と同じ結果を返しながらも、全域性・純粋性を保たなければいけない。これは各々次のように達成される。まず、ヒープを明示的に扱うことによって純粋性を得る。次いで、自然数 fuel を導入することによって全域性を得る。

4.2.1　ヒープの明示化

新しい関数ではヒープを引数として受け取り、ヒープを書き換える場合には、書き換えた後のヒープを戻り値として返すことで、ヒープに対する処理を純粋関数で行えるようにする。例えば元々次のような関数があったとする：

```
let f (storage:Storage.t) (i:Index.t) (ch:char)
  : St (* ヒープに書き込む副作用を持つ *) unit
= let buf = make_buf storage i in
  set_char buf 10 ch;
  set_char buf 11 ch
```

このとき、次のようなmodel_fが定義できる。なおstorage_modelおよびmodel_of_storageはStorage.tのモデルを扱うための型・関数で、4.3 節にて定義される：

```
let model_f
  (h:mem) (* 引数としてヒープを受け取る *)
  (storage:storage_model) (i:Index.t) (ch:char)
  : Tot (* 純粋関数である *) unit
= let buf = model_make_buf h storage i in
  let h1 = model_set_char h  buf 10 ch in
  let h2 = model_set_char h1 buf 11 ch in
  h2 (* 書き換えた後のヒープを返す *)
```

関数model_fとfが対応することは、fの型宣言において確認する：

```
(* 関数 f の型宣言 *)
val f : storage:Storage.t -> i:Index.t -> ch:char -> ST unit
  (requires (fun _ -> true))
  (ensures (fun h _ h' ->
     (* 事後条件で、関数を呼び終えた後のヒープと
        model_f の戻り値が一致することを確認する *)
     h' == model_f h (model_of_storage storage) i ch
  ))
```

これによって純粋な関数による置き換えが達成できる。

4.2.2 自然数 fuel の導入

次に再帰的な関数において、その全域性を確保することを考える。equalやload_node_fully_for_testなどの関数は、引数として渡されるノードからはじまり、再帰的に木を辿ることによって処理を行う。このとき途中のノードでDiskがあると、ディスクの読み込みを行うことでこれをViewに変換し、得られた木をさらに辿る。これは木構造を再帰的にたどっている最中に、その木の葉ノードが内部ノードに置き換わりうることを意味している。2.4 節で見たように、F* において再帰関数の停止性を示すためには、再帰呼び出しのたびに構造的に小さくなる値を用意する必要がある。しかし引数にあるノードは上記の理由からその値として使えない。ほかにも、バイト列を再帰によって読み込みつつ木を構築するparse_cell関数では、例えば自分自身のインデックスが指すノードを子ノードとしているようなノードを表す不正なバイト列が引数に

渡された場合には関数が停止しない。

そこで我々はこれらの関数に自然数fuelを引数として加え、再帰呼び出し毎にこの値を 1 ずつ減らすことによって再帰関数が停止することを保証するアプローチをとった。この方法は、Fromherz ら [13] がアセンブリコードを評価する関数eval_codeを定義する際の手法を参考にした。Plebeia においては、例えばload_node_fully_for_test関数は元々次のように定義されている（説明のために一部省略）：

```
let rec load_node_fully_for_test context n =
  let v = match n with
    | Disk i ->
        load_node context i (* i から新しくノードを読み込む *)
    | View v -> v
  in
  match v with
  | ...
  | Bud (Some n, i, hp) ->
      (* 子ノード n について再帰的に読み込む *)
      let n = load_node_fully_for_test context n in
      View (Bud (Some n, i, hp))
  | ...
```

この再帰関数呼び出し部分を次のように変更することで、再帰が必ず停止することを確認できる：

```
let rec model_load_node_fully_for_test
  h    (* ヒープも使用するため導入する *)
  fuel (* 自然数 fuel を導入する *)
  context n
= ...
  | Bud (Some n, i, hp) ->
      (* fuel を 1 減らし再帰することにより全域性を確保する *)
      let n = load_node_fully_for_test h (fuel - 1) context n in
```

```
      View (Bud (Some n, i, hp))
  | ...
```

この手法では、関数が再帰する場合はfuelが 1 以上であることを保証しなければならない。例えばmodel_load_node_fully_for_testの事前条件では、再帰的にfuelが 1 以上であることを仮定した。またmodel_parse_cell_recでは、その戻り値の型をoption viewとし、指定されたfuelで結果rが得られた場合にはSome rを、得られなかった場合にはNoneを返した。

その導入方法から言って、fuelは検証中でのみ使用するものであって、現れるのは証明中のみである。このことを静的に保証するために、我々はfuelを F* 中で単にnat型の値とするのではなく、Ghost.erased nat型の値とした。Ghost.erasedは F* の標準ライブラリが提供する関数で、検証中でのみ使用できる値を表す型を用意する。Ghost.erased nat型を持つ値は、検証中ではnatと同様に扱えるが、コード抽出されるようなコードで使用するとコンパイル時エラーになる。Ghost.erased nat型のfuelを用いると、例えばNode_storage_fst.commit_nodeの型を

```
Context.t -> Bud_cache.t -> node
  -> ST (node * Index.t * Hash_prefix.t)
```

から

```
Ghost.erased nat -> Context.t -> Bud_cache.t -> node
  -> ST (node * Index.t * Hash_prefix.t)
```

へと変更できる。Ghost.erased T型をコード抽出すると OCaml ではunit型となる。したがって OCaml 側からNode_storage_fst.commit_nodeは

```
unit -> Context.t -> Bud_cache.t -> node
  -> node * Index.t * Hash_prefix.t
```

型の関数として見える。元々のcommit_nodeとの差は、ラッパーコードにより埋める。詳細は 4.6 節にて述べる。

4.3 ディスクのモデル化

3.3 節で述べたように、Plebeia においてディスクは`Bigarray.t`を用いて memory-mapped file として表現され、その実体は`Storage.t`が持つ。実際にこれに対して書き込み・読み込みを行うときは、`Cstruct.t`型の部分配列を切り出し、これを経由して 1 バイトごとの書き込み・読み込みを行う。

`Node_storage`モジュールに関する検証を行うためには、ある部分配列へ書き込んだ後に読み込むと、先に書き込んだデータが得られるということを示さなければならない。この性質は、`Bigarray.t`が表す配列に対する 1 バイトごとの書き込みが正しく保持され、上書きされなければ後から読み込むことができるという、`Storage`や`Xcstruct`モジュールに対する仮定に立脚している。したがって、これらのモジュールを F* 上で適切にモデル化する必要がある。

`LowStar.Buffer`モジュール（`LSB`）は、F* のサブセットである Low* [14] において定義されたモジュールで、C 言語の配列のモデルである`LSB.buffer`を提供する。Plebeia の memory-mapped file を 1 次元の配列とみなせば、これを使ってモデルを構築できる。ただし`LSB`は配列の添字として符号なし 32bit 整数値を使用しているが、これは Plebeia のストレージサイズが最大 $(2^{32} - 256) \times 32 > 2^{32}$ バイトになることを踏まえると小さすぎる。そこで符号なし 64bit 整数値を添字として使用するように`LSB`を書き換え、`Buffer64`という新しいモジュールを作成し、これを用いる。

ところで、OCaml では`char`は 1 バイトの値を表す。Plebeia ではこれを利用して、バイト列を`string`型によって表現している。しかし F* の`char`は Unicode 対応しており、1 バイトとの対応がとれない。そこで OCaml の`char`を F* から使えるようにしたインタフェースファイルを作成した。以下では`Char.char`として新しく作成した`char`を参照する。またこの`Char.char`を使って OCaml の`string`に対応する`string`の代替を作成した。これは以下では`String.t`として参照する。

以上をまとめると、Plebeia で使用されている`Bigarray.t`型の値は長さ $(2^{32} - 256) \times 32$ の`Buffer64.buffer Char.char`型を持つとしてモデル化できる。また`Cstruct.t`型の値は、長さ不定の`Buffer64.buffer Char.char`型の値を持つとしてモデル化できる。各々のモデルを`storage_model`・`cstruct_model_t`と呼ぶこととし、`Storage.t`から`storage_model`への変換、及び`Cstruct.t`から`cstruct_model_t`への変換を行うための関数`model_of_storage`と

model_of_cstructの存在を仮定する。これによって、例えばmodel_make_bufは次のように書ける（一部省略・簡略化）。

```
let model_make_buf
  (h:mem) (s:storage_model) (i:Index.t)
  : cstruct_model_t
= (* インデックス i から 32 バイトの領域を、
     部分配列としてストレージ s から切り出す *)
  Buffer64.gsub (* gsub は buffer から部分配列を切り出す関数 *)
    (Stor?.b s) (* Stor?.b は s が持つ buffer を取り出す関数 *)
    (model_index2bufidx i) (* i を s 上の添字に変換する *)
    32uL (* 長さ 32 の部分配列を取得する *)

(* F* インタフェースファイルに書かれる
   make_buf のインタフェースに関する仮定 *)
val make_buf : s:Storage.t -> i:Index.t -> ST Cstruct.t
  (requires (fun h -> ...))
  (ensures  (fun h r h' -> ... ∧
     (* 返却値がモデル上で model_make_buf と等しいことを保証 *)
     model_of_cstruct r
       == model_make_buf h (model_of_storage s) i
  ))
```

LSBには配列に対する操作を記述するための述語が多くあり、証明を円滑に進めることができるという利点がある。これは特に次のような補題を証明するときに重要になる。

```
val lem : h:mem -> s:storage_model ->
          str1:String.t -> str2:String.t -> Lemma
  (requires (...))
  (ensures (
     (* i と j という異なる index を作る *)
     let h1, i = model_new_index h  s in
```

```
    let h2, j = model_new_index h1 s in
    (* i に str1、j に str2 を書き込む *)
    let buf1  = model_make_buf h2 s i in
    let buf2  = model_make_buf h2 s j in
    let h3    = model_write_string h2 str1 buf1 0 28 in
    let h4    = model_write_string h3 str2 buf2 0 28 in
    (* i から読み込んだ文字列は str1 と一致するか？ *)
    let r     = C.model_copy h4 buf1 0 28 in
    String.equal str1 r
))
```

ヒープh3においてbuf1から読み込めばstr1と一致するのは明らかである。しかし、h4はbuf2への書き込みが行われた後であり、ここにおいても読み込んだ文字列が一致することを示すためには、h3からh4への遷移においてbuf1が変更されていないことを示す必要がある。これはiとjが異なることから示唆される。これを証明するためには、Bigarray.tが表す配列のどの部分にmodel_write_stringが書き込むかを形式化し、これがbuf1と重なっていないことを示す必要がある。LSBにはloc_buffer_from_toという関数があり、これを用いてどの部分を変更したかを記述できる。これを用いてh3からh4ではbuf2が示す領域（つまりjが指す領域）のみが変化することを示し、かつiとjが異なることを示せば、補題全体を示すことができる。

4.4 equivalent_nodes と equal の対応

Node_storageモジュールには、ノードの等価性を判定するためのequal関数が定義されており、次の型を持つ：

```
Context.t -> node -> node -> Result.t unit (node * node)
```

第二引数と第三引数に比較したいノードを渡すと、それらのノードが木構造として等価かどうかを判定する。このとき indexed や hashed の等価性は考慮されないという点で、OCaml における=とは異なる意味論を持つ。渡したノードが等価であると判定された場合はResult.Ok ()が返却される。等価でないと判定された場合は、その証拠を表すノードnaとnbを用いてResult.Error (na, nb)が返却される。Node_storage.equal関数の実際の動

作は、内部で定義される補助関数によって行われている。以下ではこの関数をequal_auxと呼ぶ。

equal関数は、その名前とは裏腹に、与えられたノードの等価性を完全に調べるわけではない。具体的には、木を走査する際に、調べている二つのノードが両方とも遅延読み込み（Disk）である場合には、その読み込みを行わず、Disk iのiが等しいかのみを調べる。したがって、例えば異なるインデックスiとjに同じ内容の木が書き込まれている場合でも、以下はfalseとなる：

```
equal context (Disk i) (Disk j) = Result.Ok ()
```

equal関数のこの定義のために、equal関数は推移律を満たさない。これは、例えば次のような例を考えれば明らかである：equal context (Disk i1) (View v)かつequal context (View v) (Disk i2)かつi1 <> i2とする。このときequal context (Disk i1) (Disk i2)は成立しない。証明において等価性の推移律は重要な役割を果たすため、これは大きな問題である。

またequal_auxは末尾再帰を用いて実装されている。一般に、末尾再帰関数における検証では帰納法の仮定を直接は利用できないため、等価な非末尾再帰関数の検証と比べて複雑になりやすい。

以上の問題を解決するために、我々はequivalent_nodesという関数を新たに定義した。この関数はノード間の完全な等価性を調べる関数であり、引数のノードについて自然に再帰する非末尾再帰関数である。equivalent_nodesは次の型を持つ：

```
nat -> mem -> Context.t -> node -> node -> bool
```

第一・第二引数は 4.2 節で導入した、検証のための引数である。第四・第五引数が等価性を調べるべきノードである。戻り値の型はboolであり、引数が等価である場合、かつその場合に限りtrueを返す。equivalent_nodesは反射律・対称律を満たすほか、equalと異なり推移律も満たす。

その定義から明らかなように、equalとequivalent_nodesは等しい関数ではない。同じインデックスであれば再帰的にノードをたどっても等価であると言えるため、equalにおいて等価だと判定されればequivalent_nodesでも等価であると言える。一方でその逆は一般には言えない。そこで証明では、一方の

ノードに現れるDisk iの位置を特定し、他方の対応するノードがDisk jの場合にi = jであることを示すことで、equalにおける等価性を示した。

4.5 証明事項の定式化と証明の方針

読み込み関数としてload_nodeを用いると、証明すべき内容は次のように定式化できる。n0を任意のノードとし、hを任意のヒープとする。n0が指す木に含まれる任意のノードnについて、あるインデックスiが存在して、nがiでindexed であるならば、次が成り立つと仮定する：

- nが指す木とiから読み込める木が等価である。すなわち

```
equivalent_nodes fuel h context n (Disk i)
```

なおfuelは適切にとれるものとしている。

- nがInternal (Disk j1, Disk j2, _, _)の場合、iから読み込むノードはInternal (Disk i1, Disk i2, _, _)となり、かつこのときj1 = i1かつj2 = i2

このとき、プログラム 1 が常にtrueと評価されることを示す。

また読み込み関数としてload_node_fully_for_testを用いると、同様の仮定のもとで次のようなプログラムがtrueと評価されることを示すものとして定式化できる：

```
let open Node_storage in
let n1, i, _ = commit_node context bud_cache n0 in
let n2 = load_node_fully_for_test context (Disk i) in
equal context n0 n2 = Result.Ok ()
```

なお上記の仮定は、commit_nodeを書き込み関数として用いる場合は自然に成立することに注意が必要である。すなわち、indexed なノードを 1 つも含まないようなノードを書き込んだ結果得られたノードn1はこの仮定を満たし、そのような indexed ノードのみを indexed ノードとして含むような木もこの仮定を満たす。

証明は次のようにして行う。まずプログラム 1 のcommit_node関数の呼び出しにおいて、引数n0と戻り値n1がequivalent_nodesにおいて等価であることと、n1とDisk iが等価であることを示す。これはfuelに関する数学的帰納法

により証明できる。次いで、Disk iとn2が等価であることを示す。ここで推移律を用いると、上記の等価性をまとめてn0とn2がequivalent_nodesを満たすことが示せる。最後に、n0とn2の指す木をたどり、双方のノードがDiskの場合に、その読み込むインデックスが等しいことを示す。これによってn0とn2がequal関数の意味で等しいと言える。証明における各ステップの詳細は付録（A.1）にて述べた。

4.6 抽出された OCaml コードを用いたビルド

検証を終えた F* コードを実際に動作させるためには、F* コンパイラに備わったコード抽出機能を用いて F* コードを OCaml コードに変換しなければならない。node_storage_fst.fst から OCaml のコード抽出を行うと、Node_storage_fstモジュールの定義を行う node_storage_fst.ml ファイルが得られる。このファイルには元々 node_storage.ml ファイルに含まれていた関数が含まれているが、4.2 節での操作により、関数の API が異なる場合がある。また今回の検証に影響を及ぼさない関数などはそもそも node_storage_fst.fst に移植されていないため、当然抽出された node_storage_fst.ml にも含まれない。そこで元々の node_storage.ml が提供していた機能と node_storage_fst.ml が提供する機能との差を埋めるために、node_storage_fst.ml に定義された関数を呼び出すラッパーコードを node_storage.ml に書く。このコードは、元々 node_storage.ml に定義されていた関数について、与えられた引数をほとんどそのまま node_storage_fst.ml の対応する関数に引き渡すようなものになっている。例えばNode_storage.load_nodeは次のようなラッパーコードになる：

```
let load_node context index ewit =
  Node_storage_fst.load_node () context index ewit
```

新たに引数に増えた()は、4.2 節において導入したGhost.erased nat型のfuelによるものである。ラッパーコードは単なる OCaml コードで、それ自体に検証は行われていない。しかし全てのラッパーコードは引数を自明な形でNode_storage_fstモジュールの対応する関数に引き渡しているだけであり、その正しさはほとんど明らかであると言える。

抽出された OCaml コードとラッパーコードを、Plebeia のその他のコードと合わせてビルドすることで、最終的な実行可能バイナリを得ることができる。このように F* によって検証された OCaml コードと、そうでない Plebeia のその他のコードを合わせてビルドできることは、Plebeia の段階的な検証を可能にする。この手法では、プログラマは、全てのファイルについて検証が終わっていなくても Plebeia を動作させてテストを実行することができるため、F* への移植が間違っていないことを確かめやすいというメリットがある。

5 実験

5.1 環境

我々は次のような計算機環境で実験を行った：

CPU Intel Core i7-8700 (最大周波数 4.6 GHz)

RAM 32 GiB

OS Ubuntu 20.04.2 LTS

F* コンパイラ [15] には 2021 年 1 月 12 日現在で最新の、Git のコミット b5b1f18997 をビルドしたものを使用した。Z3 はバージョン 4.8.5 を使用した。

Plebeia [8] の元々の実装として、コミット 44532dd を使用した。また Sato [12] による検証が行われた実装としてコミット c1f8b05 を使用した。我々が検証した実装はコミット 4c3636f である。

経過時間・使用メモリの計測は全て 10 回行い、その結果を 平均値 ± 標準偏差と表記した。

5.2 結果

我々は、Sato が追加した F* ファイルを元に、Plebeia の `node_storage.ml` をいくつかの F* ファイルにポートし、証明を行った。証明全体では 17,036 行の F* ファイルを必要とした。なおこの行数には`Buffer64`のためのコードも含まれるほか、Sato が追加したファイルのうち `node_storage.ml` の検証に必要なものも含まれている。`Node_storage_fst.fst` から抽出した OCaml コード `Node_storage_fst.ml` を OCaml 側から使用するためのラッパーである新しい `node_storage.ml` は 54 行になった。

証明全体の型検査には 3462 ± 4 秒を要した。型検査中に要したメモリ量は 3100 ± 100MB であった。なお型検査はファイルごとに並列に行うことが可能

表 1　各実装が、Plebeia に付属するテストを通過するのにかかった時間（秒）

元々の実装（44532dd）	Sato の実装（c1f8b05）	我々の実装（4c3636f）
139 ± 2	126 ± 5	129 ± 6

であるが、これらの計測値は全て逐次的に行ったものである。型検査を並列に行う場合、より速く型検査が終了する一方で、要するメモリ量は増大すると考えられる。また F* には、検証を行った際にキャッシュやヒントを自動的に生成しておき、これを用いて次回以降の型検査を高速化する機能があるが、今回の計測ではこれらの機能を全く用いていない。

抽出した OCaml コードを用いて得られた実行可能バイナリは、Plebeia に元々あるテストを全て通過した。テストの通過には 129 ± 6 秒かかった（表 1）。Sato の実装と同様に、この値は元々の Plebeia の実装よりも速い。その要因として、元々の実装では動的に行っていた検査を静的に行うことができ、実行時の処理を省けたことが挙げられる。

上記の結果は、Plebeia の`Node_storage`モジュールに関わる OCaml 実装を F* で置き換えることが十分可能であることを示唆している。これによって Plebeia コードベースの安全性を高めることができるほか、実装の高速化に寄与する。

6　関連研究

HACL* [16] は、複数の暗号学的プリミティブを含むライブラリであり、Low* [14] を用いて実装されている。Low*は、C 言語の意味論に合わせて設計された F* のサブセットであり、KreMLin というコンパイラを用いることで Low*コードから C コードを抽出することを可能にした。これによって HACL* では、メモリ安全性・ある種のサイドチャネル攻撃の軽減・仕様と実装の対応などが検証された暗号学的ライブラリを C 言語で提供した。このライブラリは証明による安全性を備えながらも、従来の純粋な C 言語ライブラリと同程度か、それ以上の速度で動作する。HACL*は現在までに Firefox [17] や Tezos においてライブラリとして使用されている。

Bhargavan ら [7] は Ethereum 上で動作するスマートコントラクトを解析・検証するためのフレームワークを F* を用いて構築した。彼らは Ethereum

Virtual Machine 上で動作するバイトコードの一部と、そのバイトコードにコンパイルされる Solidity というプログラミング言語の一部を F* のサブセットとして表現し、F* の型システム上でそれらの検証を行うことを可能にした。彼らが暗号通貨上で動作するスマートコントラクトを検証しているのに対して、我々は暗号通貨そのもののシステムを F* を用いて検証した。

Pîrlea ら [6] はブロックチェーンに基づいた分散型合意形成プロトコルについて Coq を用いて形式検証を行った。彼らは、プロトコルが依存する構成要素に関する仮定を明確にし、小ステップ操作的意味論を用いてプロトコルの実行をモデル化した上で、その一貫性がネットワーク全体で維持されることを証明した。彼らの Coq による実装は、検証を容易にするために意図的に最適ではないため、コード抽出によって得られる OCaml コードを製品において使用することは現実的でない。一方で我々は、Tezos において使用される予定のコードそのものに検証を行った。その性能は、元々の OCaml 実装と同等か、むしろ実行時の検査を省略できるために速くなる。

Sato [12] は F* を用いて Plebeia の Plebeia tree に対する操作に形式検証を行った。具体的には、操作される木構造が Plebeia tree の不変条件を満たしていることと、操作の結果が正しいことを証明し、Plebeia の OCaml コードである `node.ml` と `cursor.ml` を `node.fst` と `cursor.fst` で置き換えた。本研究はこの研究に続くものであって、本研究で使用した F* コードは Sato が実装したこれらの F* ソースファイルに基づいている。

7 結論

本研究では、暗号通貨の一つである Tezos での使用を見込まれているストレージシステム Plebeia において、データ永続化処理を担う`Node_storage`モジュールの実装を F* を用いて形式的に検証した。`Node_storage`モジュールは、Plebeia の内部データである Plebeia tree をバイト列に変換してディスクに書き込む関数と、ディスクからバイト列を読み込んで Plebeia tree を再構築する関数からなる。我々は前者の関数を使ってディスクへの書き込みを行った後、後者の関数を用いてディスクから読み込んだ際に、書き込んだデータと読み込んだデータが一致するための十分条件を明らかにし、実装の正しさを検証した。

F* は OCaml によく似た関数型プログラミング言語であり、プログラムを検

証するための機能が多数盛り込まれている。我々はまず、OCaml で書かれた`Node_storage`モジュールを F* にポートし、次いで F* 上で証明を行い、その後 F* のコード抽出機能を用いて F* コードを OCaml コードに変換することで、形式的に検証された`Node_storage`モジュールを得た。

我々は検証のために 17,036 行の F* コードを用いた。検証の実行にはおよそ 3,462 秒を要した。コード抽出によって得られた OCaml コードを用いてビルドしたバイナリは、元々 Plebeia に存在した全てのテストを通過し、その動作速度は元々の実装と同程度に速かった。

`Node_storage`モジュールには、ノードを再帰的にたどって読み込むための`load_node_fully_for_test`関数の他に`load_node_fully`という関数がある。これを用いた検証は、`load_node_fully`の実装が複雑なため行っておらず、今後の課題である。また今回の検証は Plebeia のコミット `44532dd` を元に行っているが、これは 2021 年 1 月 21 日時点で 7 ヶ月前のコードベースであり、かなり古い。Plebeia の最新版への証明の追従も今後の課題であると言える。

謝辞

本研究の機会を頂き、多くのご指導を賜りました末永幸平准教授・五十嵐淳教授に深く感謝いたします。また基礎となる F* コードを頂いたほか多くの助言を賜った佐藤聡太氏と、Tezos・Plebeia の実装について解説頂いた古瀬淳氏に感謝いたします。最後に、本報告書の原稿に目を通して頂いた和賀正樹助教をはじめ、多くの助言を頂いた研究室の皆様に深く感謝いたします。

参考文献

[1] Nakamoto, S.: Bitcoin: A Peer-to-Peer Electronic Cash System (2009). `https://bitcoin.org/bitcoin.pdf` (Accessed on January 10th, 2021.).

[2] Wood, G.: Ethereum: A secure decentralised generalised transaction ledger (2014). `https://ethereum.github.io/yellowpaper/paper.pdf` (Accessed on January 10th, 2021.).

[3] Siegel, D., Paulsen, J. H., Michelet, A., Wiedmann, P. and Tacke, L.: The DAO Chronology of a daring heistand its resolution (2016). `https://www2.deloitte.com/content/dam/Deloitte/de/Documents/Innovation/Deloitte_Blockchain_Institute_Whitepaper_The_DAO.`

pdf (Accessed on January 10th, 2021.).

[4] 独立行政法人 情報処理推進機構ソフトウェア・エンジニアリング・センター: 「形式手法適用調査」報告書 (2010). https://www.ipa.go.jp/sec/softwareengineering/reports/20100729.html (Accessed on January 15th, 2021.).

[5] Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M. and Deardeuff, M.: How Amazon Web Services Uses Formal Methods, *Commun. ACM*, Vol. 58, No. 4, p. 66–73 (2015).

[6] Pîrlea, G. and Sergey, I.: Mechanising blockchain consensus, *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, Association for Computing Machinery, pp. 78–90.

[7] Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N. and Béguelin, S. Z.: Formal Verification of Smart Contracts: Short Paper, *PLAS@CCS 2016* (Murray, T. C. and Stefan, D.(eds.)), ACM, pp. 91–96 (2016).

[8] DaiLambda, Inc.: dailambda / plebeia · GitLab. https://gitlab.com/dailambda/plebeia (Accessed on January 19th, 2021).

[9] DaiLambda, Inc.: Plebeia in Tezos. https://www.dailambda.jp/blog/2020-05-11-plebeia/ (Accessed on January 19th, 2021).

[10] Goodman, L.: Tezos—a self-amending crypto-ledger: White paper (2014). https://www.tezos.com/static/papers/white_paper.pdf (Accessed on January 10th, 2021.).

[11] Swamy, N., Hriţcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.-Y., Kohlweiss, M., Zinzindohoue, J.-K. and Zanella-Béguelin, S.: Dependent Types and Multi-Monadic Effects in F*, *SIGPLAN Not.*, Vol. 51, No. 1, p. 256–270 (2016).

[12] Sato, S.: Verification of a Merkle Tree Library Using F*, Master's thesis, School of Informatics and Mathematical Science, Faculty of Engineering, Kyoto University (2021).

[13] Fromherz, A., Giannarakis, N., Hawblitzel, C., Parno, B., Rastogi, A. and Swamy, N.: A Verified, Efficient Embedding of a Verifiable Assembly

Language, *Proc. ACM Program. Lang.*, Vol. 3, No. POPL (2019).

[14] Protzenko, J., Zinzindohoué, J.-K., Rastogi, A., Ramananandro, T., Wang, P., Zanella-Béguelin, S., Delignat-Lavaud, A., Hritcu, C., Bhargavan, K., Fournet, C. and Swamy, N.: Verified Low-Level Programming Embedded in F*, *PACMPL*, Vol. 1, No. ICFP, pp. 17:1–17:29 (2017).

[15] Swamy, N. and Microsoft Research: FStarLang/FStar: Verification system for effectful programs. `https://github.com/FStarLang/FStar` (Accessed on January 28th, 2021).

[16] Zinzindohoué, J. K., Bhargavan, K., Protzenko, J. and Beurdouche, B.: HACL*: A Verified Modern Cryptographic Library, *CCS 2017* (Thuraisingham, B. M., Evans, D., Malkin, T. and Xu, D.(eds.)), ACM, pp. 1789–1806 (2017).

[17] Beurdouche, B.: Verified cryptography for Firefox 57. `https://blog.mozilla.org/security/2017/09/13/verified-cryptography-firefox-57` (Accessed on January 28th, 2021).

[18] De Moura, L. and Bjørner, N.: Z3: An Efficient SMT Solver, *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, Berlin, Heidelberg, Springer-Verlag, p. 337–340 (2008).

付録

A.1 証明の概略

読み込み関数としてload_nodeを用いる場合の証明の概略は次のようになる。なおload_node_fully_for_testを用いる場合も同様に証明できる。また、厳密な証明は全て F* において形式化され、F* コンパイラによって自動的に検証されることに注意が必要である。

評価するべき OCaml コードは次のような F* コードと等価である。

```
let h1, n1, i, _ =
  model_commit_node fuel h0 context bud_cache n0 in
let n2 = View (model_load_node fuel h1 context i) in
let fuel' = necessary_fuel_of_equal_aux_precondition
              fuel 1 fuel h1 context n0 n2 in
model_equal fuel' fuel h1 context n0 n2 = Result.Ok ()
```

ただしここでnecessary_fuel_of_equal_aux_precondition関数はequal_auxが必要とするfuelを計算するための関数である。equal_auxとequivalent_nodesでは木の辿り方が異なるため、必要とするfuelも異なる。引数のノードがequivalent_nodesを満たす場合には、equal_auxが必要とするfuelを計算することができる。

まずequivalent_nodes fuel h1 context n0 n1を、fuelに関する数学的帰納法で示す。

- n0がDisk _であるときは、commit_nodeとequivalent_nodesの定義から明らかである。
- n0が indexed なView _であるときは、commit_nodeの定義よりn0 = n1である。equivalent_nodesは反射律を満たすため、成立する。
- n0が indexed でないView vであるときは、vの種類によって場合分けする。
  - Leaf (value, _, _)のとき、n1は同じvalueを持ったLeafとなる。したがって成立する。
  - Bud (None, _, _)のとき、n1は同様にNoneを持ったBudとなる。したがって成立する。
  - Bud (Some n, _, _)のとき、nについてcommit_node_auxの再帰的な

呼び出しが行われる。これによって子ノードが書き込まれる。この呼び出しの結果得られるノードをn'とし、ヒープをh'とすると、帰納法の仮定からequivalent_nodes (fuel - 1) h' context n n'が成り立つ。vそのものを書き込む際にヒープが変化するが、この変化はnとn'の等価性を保つ。ここでn1はn'を持ったBudとなるため、equivalent_nodes fuel h' context n0 n1が成立する。したがって成り立つ。

- Extender・Internalについても Budと同様に成立する。

次にequivalent_nodes fuel h1 context n1 (Disk i)を、fuelに関する数学的帰納法で示す。

- n0がDisk _であるときは、commit_nodeとequivalent_nodesの定義から明らかである。
- n0が indexed なView _であるときは、仮定より成立する。
- n0が indexed でないView vであるときは、vの種類によって場合分けする。
  - Leaf (value, _, _)のとき、commit_nodeはiとして、この葉を書き込んだ先の領域を指す index を返す。この index をparse_cellに渡すとn1と同様にvalueを持ったノードを得られることが分かる。よって成立する。
  - Bud (None, _, _)のとき、commit_nodeはiとして0を返却する。これは特殊な index であって、Bud (None, _, _)を一意に表す。したがってこの index をparse_cellに渡すとn1と同様にBud (None, _, _)を得られると分かる。よって成立する。
  - Bud (Some n, _, _)のとき、nについてcommit_node_auxの再帰的な呼び出しが行われる。これによって子ノードが書き込まれる。この呼び出しの結果得られるノードをn'、インデックスをi'とし、ヒープをh'とすると、帰納法の仮定からequivalent_nodes (fuel - 1) h' context n' (Disk i')が成り立つ。これはvを書き込む操作によるヒープ変化を経ても成立する。ここでn1はn'を持ったBudとして構成され、またiからの読み込みではi'から読み込んだノードを子ノードとして持つBudが構成される。従って成り立つ。
  - Extender・Internalについても Budと同様に成立する。

その後equivalent_nodes fuel h context (Disk i) n2を示す。これはequivalent_nodesとload_nodeの定義をみると明らかである。

以上から、equivalent_nodesの推移律を用いてequivalent_nodes fuel h1 context n0 n2が言える。

次に、n0とn2の各ノードを根から対応させつつ同時にたどることを考える。双方がDiskのとき、その読み込むインデックスが等しいことを示す。これは、n2がload_nodeの呼び出しによって得られたことと、load_nodeが呼び出すparse_cellがInternalの子ノードにおいてのみDiskを使用することを考慮すると、n0に含まれるInternalの子ノードで現れるDisk iが、対応するn2のDisk jについてi = jを満たせば良い。そこで、commit_nodeによって書き込まれたiをjとしてparse_cellが取り出すことを示す。これはparse_cellのfuelに関する数学的帰納法によって示すことができる。なおcommit_nodeはindexed なノードの場合はキャッシュした index をそのまま返すため、ここで仮定を用いる必要がある。

以上より、fuel'に関する数学的帰納法を用いることでequal fuel' fuel h1 context n0 n2が言える。

A.2 抽出されたコードの最適化

OCaml コードを F* に移植し、コード抽出によって検証された OCaml コードを得る場合には、元々の OCaml コードとほぼ等価なコードを得ることが多い。そのため、検証による性能劣化は起こりにくい。しかし、場合によっては、F* コンパイラが挟むラッパーコードやその他要因によって、速度が遅くなる場合がある。

速度が遅くなった場合には、プロファイラを用いてその原因を探ることができる。OCaml コンパイラはバージョン 4.09.1 から gprof のサポートを落としてしまったため、代わりに perf コマンドを用いる[1)]:

```
# 実際にプログラムを実行して、プロファイルを取る
$ perf record --call-graph=dwarf -- ./your-program.exe arguments
# プロファイルの結果を見る
```

1) https://github.com/ocaml-bench/notes/blob/master/profiling%5Fnotes.md も参考のこと。

```
$ perf report
```

環境によっては sudo を前置する必要がある。

FStar.UInt8.uint_to_tなどの、F* における machine integer を OCaml の integer に変換する関数は、大量に呼び出すと抽出後コードの速度低下を招く。これらの関数は、例えば F* 中で255uyのように machine integer のリテラルを用いると、コード抽出後の OCaml コードに挿入されるため問題になる。これを防ぐためには、使用するリテラルをトップレベルで次のように定義しておき、関数内部ではこれを用いる。

```
let uint8_255 = 255uy (* トップレベルで定義する *)
(* 以降では 255uy の代わりに uint8_255 を用いる *)
```

HyperStack.ST.getもまた、大量に呼び出すと抽出後コードの速度低下を招くため、不必要に呼ぶことは避ける必要がある。