証明駆動開発の現実的な開発プロジェクトへの適用に向けて

山本 晃治[1,a)]　宗像 一樹[1,b)]

概要：形式的証明やそれに基づく証明駆動開発は工数増加の懸念から採り入れられないことが多い．しかし今回実験の過程で実プロジェクトで使用中のプログラムに対して，テストで見逃されていた潜在バグを発見した．これは形式的証明の有用性の一端を示すものである．本稿では，実業務での適用を念頭に，証明駆動開発をプログラムの動作保証に活用するための方法を提案する．
実業務で稼働させるアルゴリズムに対して，その性質を Coq で形式的に証明し，証明済みプログラムを，通常の実業務でよく用いられる非関数型プログラム言語に変換するモジュールについて説明する．また関数型プログラム言語である Coq から非関数型プログラム言語 Python への変換結果のコード実行時のメモリ使用量の悪化を防ぐ方策を提示し，10 倍程度の悪化にまで抑えられることを示す．

キーワード：証明駆動開発，形式的定理証明，Coq，コード生成，非関数型言語，Python

1. はじめに

テストを用いた一般的な開発では，記述するプログラムが所望の性質を満たすことをテストで確認することで，プログラムの動作を保証する．このテスト作業を形式的な定理証明作業に置き換えると，証明作業の手間と引き換えにテストで起こりうる確認漏れを防げる可能性がある．例えば，図 1 の Python3 関数 diffSortedLists は，2 つのソート済みリストの要素を比較するコードであり，バグが混入している [*1]．

今回の手法の実験中に手組みコード（図 1）と証明済みのコードの実行結果の差異から，このバグを発見した．単体テスト 17 項目を通過し 9 ヶ月以上業務で使われてきたコード [*2] からバグが発見できたことは，定理証明作業が確認漏れをテストより防ぎやすい可能性を示している．

全ての性質確認を証明作業で行うことは現時点では困難だとしても，一部なら，性質確認をテスト作業から証明駆動開発 [1], [2] と呼ばれる，テスト駆動開発のテスト作業を定理の形式的証明作業に差し替えた開発形態への置き換え可能と考える．図 2 は証明駆動開発の作業ステップの一例である．プログラムを定理証明支援器が理解できるプログラム言語で記述しながら，並行してプログラムが満たすべき性質を定理として記述し，定理を証明する．その後，定理証明済みのプログラムを実際の開発で用いるプログラム言語に変換する，というステップを踏む．

```
def diffSortedLists(l1, l2):                                    1
    l2fresh=True; l2end=False; iter2=iter(l2)                   2
    #e2toEmit = None                                            3
    for e1 in l1:                                               4
        #if e2toEmit and (e1 > e2toEmit):                       5
        #    yield (1, e2toEmit)                                6
        #e2toEmit = None                                        7
        while not l2end and l2fresh or e1 > e2:                 8
            try:                                                9
                e2 = next(iter2)                                10
                if l2fresh: l2fresh = False                     11
                if e1 > e2: yield (1, e2)                       12
            except StopIteration:                               13
                e2 = None; l2end = True; break                  14
        if l2end or e1 < e2:                                    15
            yield (-1, e1)                                      16
            #if not l2end: e2toEmit = e2                        17
        else: yield (0, e1)                                     18
    if l2end: return                                            19
    if not l2fresh and e1 < e2: yield (1, e2)                   20
    for e2 in iter2: yield (1, e2)                              21
```

図 1　テストで混入バグを見つけられなかったコード
Fig. 1　A code having a bug that cannot be found with testing.

[1] 富士通研究所
Fujitsu Laboratories, Kawasaki, Kawanaga 211–8588, Japan
[a)] yamamoto.kouji@jp.fujitsu.com
[b)] munakata.kazuki@jp.fujitsu.com

[*1] バグの内容は，例えば l1=[1,2,4], l2=[1,3,4] が入力に与えられると [(0,1), (-1,2), (1,3), (0,4)] が返されるべきなのに対し，(1,3) が抜け落ちるというものである．修正には図でコメントアウト行のコメントを外す必要がる．

[*2] 298 日間で少なくとも 4904 回は呼び出され，その結果が GUI を通して人目に触れてきたコードである．

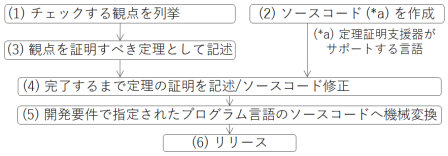

図 2 証明駆動開発の流れ
Fig. 2 Steps for proof driven development.

ただし，証明の記述が正しいことの判断を人手によらず機械的に行える（以降，証明の機械的確認とよぶ）ことが望ましく，プログラム言語の変換前後でコードの振舞いが観測等価であると保証（以降，コード等価性保証とよぶ）できることが望ましい．

前者の証明の機械的確認を行える定理証明支援器には Coq[3] や Isabelle[4] がある．このうち Coq は，証明した対象プログラムを別のプログラム言語に変換する機構（Coq の文書ではプログラム抽出とよんでいる）を備えており [5], [6]，コード等価性保証も証明されている．さらに変換先プログラム言語を追加できる構造となっている [*3]．

このことから以降は Coq を使う前提で議論を進める．

証明駆動開発を前述（図 1）のプログラムの開発に適用する場合，我々にとって最も大きな障壁は，証明時に用いるプログラム言語から実開発に用いるプログラム言語（Python3）への変換が用意されていない点であった．

しかし，定理証明支援器 Coq に備わったプログラム言語抽出機構を利用すれば，プログラム変換は可能なはずである．そこで Python3 用の変換モジュールを作り，次の 3 点に関して実用に耐えうるかを確認した:

RQ1: 業務ツール中のアルゴリズムを変換可能か
RQ2: 証明駆動開発で作った証明用言語コードが実プログラミング言語に変換後バグなく動作するか
RQ3: 変換されたコードは，手組みで作成したコードに比べてどの程度のメモリ使用量増加や速度低下があるか

本稿の貢献は次の 4 点である:

(1) 実業務で稼働させるアルゴリズムに対してその性質を形式的に証明した．
(2) 証明支援器 Coq の記述から非関数型言語である Python3 へ変換する抽出モジュールを作成した．
(3) 関数型プログラム言語である Coq から手続き型プログラム言語である Python3 への変換について，メモリ使用量増加を抑える方策を提案した．
(4) 実業務で稼働させるアルゴリズムのバグを，形式的証明を元に発見した．

本稿の構成は次のとおり．第 2 節で形式的証明をとりまく背景を述べたのち，第 3 節で証明駆動開発の 1 アプローチと，証明例，証明済みコードの変換抽出モジュール，メモリ使用量等を抑える方策を説明する．第 4 節でこの評価を行い，第 5 節，第 6 節で関連研究とまとめを行う．

2. 背景

2.1 形式的証明の開発への活用

一定の規則（例えば書き換え規則）を利用して機械的に真偽を確認できるよう一定の書式で証明を記述することを，本稿では形式的証明と呼ぶ．

形式的証明をプログラム開発に活用するアプローチは数多く議論されている．例えば CompCert[9] は，仕様どおりに実装されていることを証明で確認したうえでリリースする手順を採っている．

2.1.1 証明駆動開発

テスト駆動開発は先に通すべきテストを記述してそのテストが通るようにプログラムを実装し改良する．このアナロジーで，証明駆動開発は先に動作保証が必要な所望の性質を定理として記述してその定理が証明できるようにプログラムを実装し改良する．

証明駆動開発についての言及は多い [1], [2] が，多くは関数型プログラム言語でのプログラム開発に限定されている．これを利用者数や利用機会の多い非関数型プログラム言語に広げることが本稿の目的のひとつである．

2.2 プログラム抽出

証明されたコードは一般に，開発要件とは異なるプログラム言語で記述されるため，プログラム抽出により，開発に用いるプログラム言語に変換する必要がある．

2.2.1 Coq の持つプログラム抽出機能

Coq で扱える書式のプログラムから，任意のプログラム言語の書式に変換したプログラムを抽出する機能が Coq に内蔵されている [6]．元の Coq 用プログラムを，プログラム言語 ML のサブセットである MiniML の構文木に変換し，この構文木を入力に個々の言語仕様にあわせ pretty print するモジュールを追加する方式を採っている．

MiniML の構成要素（MiniML の抽象構文木の項）は変数参照，関数定義など関数型プログラム言語でも手続き型プログラム言語でも現れる概念を含む．さらに関数へのパラメタの部分適用，let in 式といった，関数型プログラム言語のみで一般的な概念も含む．また最新の Coq 8.5 の実装までに，当初の論文 [5] では現れなかった，（再帰データ型の）パターンマッチの概念が追加されたもようである．Python3 への変換モジュールでは，これらの関数型プログラム言語の概念に対応する必要がある．

3. 証明駆動開発

定理証明は確認漏れが起こりにくいが，テストは幅広い観点で性質を確認できる [*4]．確認漏れが起こりにくいこ

[*3] 実際に変換先プログラム言語を追加した報告もある [7], [8]．ただし，コード等価性保証は別途必要．

[*4] いずれの手段でも，確認する性質の観点の列挙やその妥当性チェックは人手で行うしかない．

入力: ソート済みの 2 つのリスト．リスト要素の型は大小比較と等価判定が可能な任意の型
出力: 次に示す要素からなる 2 つ組のリスト．ただし，リスト要素は 2 つ組の第 2 要素でソートされていること:
第 1 要素 入力のどちらのリストに存在するかを表す記号 –1, 0, +1．ただしそれぞれ 1 つ目のリストのみ，両方のリスト，2 つ目のリストのみに存在することを表す．
第 2 要素 いずれかの入力リストに現れた要素．

［例］入力: $(0,1),(1,2)$，出力: $(-1,0),(0,1),(+1,2)$

図 3 ソート済みリストの差分抽出アルゴリズムの仕様
Fig. 3 A specification to etract difference b/w. two sorted lists.

とが幅広い観点での確認より優先されると判断するなら，動作保証手段として形式的証明を開発に活用すべきと考える．我々は，従来の開発手順の一部を証明駆動開発作業に差し替える形での開発手順を採った．

3.1 証明駆動開発のステップ

決めた設計を元に詳細設計しコーディングしその結果をレビューとテストで確認する作業を，特定のモジュールに限って，定理証明支援器用の言語でコーディングする作業（図 2 の (2)），モジュールの性質を定理として記述する作業（図 2 の (3)），定理を証明する作業（図 2 の (4)）の 3 作業に切り替える．なおこの 3 作業はその順に作業を完了させるのではなく，いずれかの作業の途中で他の作業の成果物の誤りに気づき修正することを繰り返す．

3.2 定理証明支援器用の言語でのコーディング

図 3 は，冒頭に挙げたコードの基本設計/外部仕様である．この図 3 の仕様に対するコードの記述例を図 4 に示す．

図 4 の 6 行目から始まるアルゴリズム本体では，リストの 2 つ組を個々のリストに分解し（9 行目），1 つ目のリストが空（10〜13 行目）か非空（14〜22 行目）かをパターンマッチを使って場合分けし，それぞれの場合について 2 つ目のリストが空（11 行目，14 行目）か非空（12 行目，16〜21 行目）かで場合分けする．両方のリストが非空の場合は 2 つのリストの先頭要素が一方より小さい（18 行目）か等しい（20 行目）か大きい（21 行目）かで場合分けする．

7, 8 行目はこの関数 dSL の停止性の根拠を明記したもので，再帰的に呼び出されるたびに 2 つのリストの長さの和が減少することを Coq に伝えている．ここでは省略しているが，23 行目のあとに実際に減少することの証明を記述する必要がある．

3.3 コードの性質の定理記述

記述したコードの性質を表す定理を記述すると図 5 のようになる．ここでは「入力 l1 のみ」についての定理 1 つのみを挙げたが，「入力 l2 のみ」「両方」「いずれの入力

```
(* 入力のどちらのリストに存在するか表す記号の定義 *)                    1
Inductive difftype : Type :=                                          2
  | d_minus : difftype | d_zero : difftype                            3
  | d_plus : difftype.                                                4
(* 差分抽出アルゴリズム本体の定義 *)                                    5
Function dSL (lp: list nat * list nat)                                6
  {measure (fun (lp: list nat * list nat) =>                          7
    length (fst lp) + length (snd lp)) lp} :=                         8
  let (j, l) := lp in match j with                                    9
    | nil => match l with                                             10
        | nil => nil                                                  11
        | b::m => (d_plus, b) :: dSL (j, m)                           12
      end                                                             13
    | a::k => match l with                                            14
        | nil => (d_minus, a) :: dSL (k, l)                           15
        | b::m =>                                                     16
          if ltb a b                                                  17
          then (d_minus, a) :: dSL (k, l)                             18
          else if eqb a b                                             19
             then (d_zero, a) :: dSL (k, m)                           20
             else (d_plus, b) :: dSL (j, m)                           21
      end                                                             22
  end.                                                                23
(* 仕様の入力に合わせるためのラッパ関数の定義 *)                        24
Definition diffSortedLists l1 l2 := dSL (l1,l2).                      25
```

図 4 定理証明支援器 Coq で読み込み可能なコード
Fig. 4 A code written in Coq acceptable language.

リストにも現れない」のバリエーションがある．また前提と結論を逆にしたバリエーション，例えば図 5 に対しては「d_minus つきで出力される要素は入力 l1 のみに含まれる」の形の定理がある．計 8 個の定理を用意し，この入出力値の関係に関する対象モジュールの性質を網羅的に記述した．

```
Theorem first_only_outputs_minus: (* 性質1a:入力                      1
   l1 のみに現れる要素は d_minus つき出力となる *)                     2
  forall l1 l2, forall e,                                             3
    sorted l1 -> sorted l2 ->                                         4
    e \in l1 -> ~ e \in l2 ->                                         5
    (d_minus, e) \in (diffSortedLists l1 l2).                         6
```

図 5 コードの性質を表す定理 (抜粋)
Fig. 5 An excerpt of theorems representing code properties.

3.4 プログラムの性質の証明

列挙した定理それぞれに対する証明を行う．その記述は例えば図 5 の定理の場合，図 6 のようになった．

今回は証明の主要部分を完成させることを優先し，そのための基本的な補題の証明を簡便にするため，図 3 の仕様ではリストの要素の型を「大小比較と等価判定ができる

```
Proof.  intros l1 l2 e S1 S2 in1 in2.
induction l1 as [|c' l1'].
Case "l1=nil".  contradiction.
Case "l1=c'::l1'".  induction l2 as [|d' l2'];
 unfold diffSortedLists; rewrite dSL_equation.
 SCase "l2=nil".  destruct in1 as [in1 | in1].
  SSCase "e=c'".  left. rewrite in1. reflexivity.
  SSCase "e\in l1''".
   right.  apply Sorted_inv in S1 as [S1' _].
   apply (IHl1' S1' in1).
 SCase "l2=d'::l2'".  assert (S1' := S1).
  apply Sorted_inv in S1' as [S1' R1'].
  apply Decidable.not_or in in2 as [in2 in2'].
  destruct (lt_eq_gt c' d') as [[Hc|Hc]|Hc].
  SSCase "c'<d'".  rewrite (lt_ltb _ _ Hc).
   destruct in1 as [in1 | in1].
    (*e=c'*) left. rewrite in1. reflexivity.
    (*e\in l1'*) right. apply (IHl1' S1' in1).
  SSCase "c'=d'".  rewrite (eq_ltb _ _ Hc).
   rewrite (eq_eqb _ _ Hc).
   right.  rewrite <- Hc in in2.
   apply not_eq_sym in in2.
   rewrite (In_tail _ c' _ in2) in in1.
   apply (sl_minus_tail_2 _ d' _ _ S1' S2).
   apply (IHl1' S1' in1).
  SSCase "c'>d'".  rewrite (gt_ltb _ _ Hc).
   rewrite (gt_eqb _ _ Hc).
   right.  assert (S2' := S2).
   apply Sorted_inv in S2' as [S2' R2'].
   apply (IHl2' S2' in2').  intros _ in3.
   apply (sl_minus_tail_2 _ d' _ _ S1' S2).
   apply (IHl1' S1' in3).                    Qed.
```

図 6 前述（図 5）の定理の証明
Fig. 6 A proof for the corresponding theorem in Fig. 5.

任意の型」としていたところを自然数型 (ペアノ算術の自然数）に限定し [*5]，その場合について証明した．元の型は Python3 上の文字列型 (str) であり，せめて文字列型に対する証明を行うほうが適切である．しかし次の 2 点から，文字列型に対する証明に変えるのは容易と考えている．

(1) 文字列値は Unicode 文字のリストであり，辞書式に並べることで自然数と 1 対 1 に対応づけられる．
(2) 自然数に関する補題のうち証明で使っているのは，大小比較に関する補題と，次の「2 つの値は等しいか等しくないかのいずれかである」という補題のみである．
Lemma eq_dec: $\forall n\ m$: nat, $\{n = m\} + \{n <> m\}$.
いずれも証明することは困難ではない．

さらに (2) の補題を「大小比較と等価判定ができる任意の型」に対して予め証明しておくことで，今回の証明全体を，図 3 の仕様で提示した型での証明に変えることも困難ではない．

また，図 6 で apply （補題の適用）や rewrite （補題で証明された等価性を使った書き換え）を行っている行で用いている補題の多くは，今回新たに証明したものである．総数 53 個の補題を証明した（記述量は計 1376 行）．この補題を使い 8 個の定理を証明した（計 580 行）．

3.5 Python3 プログラム抽出モジュール

Coq で扱える図 4 のような形式のプログラムを Python3 プログラムに変換するモジュールを開発した．

MiniML の抽象構文木の項のうち次の種類には対応した: 変数参照，関数適用，関数定義，グローバル参照，パターンマッチ，再帰関数，（関数定義の中で現れる）let in 式．

次の種類の項には未対応である: （関数定義の外で現れる）let in 式，タプル，例外，（効率のよいコード生成のための）変換パターンのユーザ定義（Extract Inductive）．

Coq 上でのプログラム記述は OCaml に近く関数型言語との親和性が高い形式である．標準でプログラム抽出できる変換先言語も OCaml, Haskell, Scheme の 3 つの関数型言語である [*6]．

Python3 のような非関数型プログラム言語に変換する場合には，変換元の MiniML に現れる概念が変換先の言語仕様に現れないケースがある．Python3 の場合，再帰データ型（図 4 の 2〜4 行目）とパターンマッチ（図 4 の例えば 10〜13 行目），let in 式（図 4 の 9 行目），関数に対するパラメタの部分的な適用 [*7] が該当する．その対応のため，変換ルーチンに情報を付加する方式（第 3.5.1 節）と，仕様の際を吸収するライブラリを用意する方式（第 3.5.2 節）を採った．

3.5.1 抽象構文木からの変換時の情報付加

パターンマッチのうち再帰データ型を扱う部分について，Haskell など関数型言語では再帰データ型のコンストラクタのパラメタに現れた仮引数を以降の文脈で参照する．非関数型言語，少なくとも Python3 では次節第 3.5.2 節のような差異吸収ライブラリが設計できないと考えている．そこで仮引数名をパターンマッチ対象の（再帰データ型）インスタンスの内部情報を参照する処理に置換する方法を採る．Python3 用抽出ルーチンでは，MiniML 抽象構文木を深さ優先トップダウンで辿る際に，構文木の根側のノードから仮引数名と参照処理コードの対応表を伝播させる．

主だった実装内容を図 7 に示す．構文木のノード訪問ごとに呼び出される関数 pp_expr に，対応表 (subst) を引

[*5] このことは図 6 中には現れておらず，読み取れない．

[*6] バージョン 8.5 になって JSON 形式の出力が増えた．MiniML の抽象構文木をそのまま抽出する目的で用意されたと考える．

[*7] def add(x, y): return x + y のように，複数のパラメタが渡される関数に対して，一部のパラメタだけを渡した結果を得て，以後のこりの引数だけをパラメタとして受け取る関数のように扱うこと．この add の場合，例えば x に値 -1 を渡し，以後デクリメントする 1 引数関数のように扱うこと．

数に追加しており，変数参照を出力する際は対応表にしたがった参照処理コードを出力する（7, 8 行目）．対応表は構文木上の部分木内のみに伝播する．ただし，再帰関数の定義の際は兄弟の部分木にまで参照処理コードを伝播する必要があるため，専用の対応表を参照型の変数で用意し（3 行目）破壊的代入で対応表を更新・伝播する（17, 18 行目）．

```
(* 変数参照 (for Python3) *)
module Subst = Map.Make(Id)
let subst_r = ref Subst.empty
let pr_id_subst id subst =if Subst.mem id subst
  then Subst.find id subst else (pr_id id)
let rec pp_expr env subst args = function
| MLrel n -> let id = get_db_name n env in
  apply (pr_id_subst id subst)
(* 比較: 変数参照 (for Haskell) *)
let rec pp_expr env args = function
| MLrel n -> let id = get_db_name n env in
  apply (pr_id id)
(* 再帰関数定義 (for Python3) *)
and pp_fix env subst i (ids,bl) args =
(prvect_with_sep (fun () -> str "")
  (fun (fi,ti) ->
    (subst_r := push_subst !subst_r
        [(fi, str "defi." ++ (pr_id fi))];
     pp_function env !subst_r (pr_id fi) ti))
  (Array.map2 (fun a b -> a,b) ids bl)) ++
pp_apply (pr_id_subst ids.(i) !subst_r) args
(* 比較: 再帰関数定義 (for Haskell) *)
and pp_fix env i (ids,bl) args =
(prvect_with_sep (fun () -> str "")
  (fun (fi,ti) ->
     pp_function env (pr_id fi) ti)
  (Array.map2 (fun a b -> a,b) ids bl)) ++
pp_apply (pr_id ids.(i)) false args
```

図 7 置換情報の伝播コードの概要（説明に関わらない部分は省略）
Fig. 7 A excerpt code to propagate substitution information.

3.5.2 非関数型言語への変換のための差異吸収ライブラリ

再帰データ型に対しては，図 8 の Python3 クラス IndType を用意し，これを継承して再帰データ型の型の定義（図 4 の 2〜4 行目の場合，2 行目の difftype の定義）を実装したクラスを作り（図 9 の 1, 2 行目），再帰データ型のコンストラクタの定義 (図 4 の 3, 4 行目の定義) は，作ったクラスをさらに継承したクラスを作って対応した（図 9 の 3, 4 行目）．再帰データ型の参照時には，コンストラクタに渡したパラメタ（図 4 では現れない）等の必要な情報がクラス IndType（図 8）に格納されているので，これを取り出すコードへ変換する（第 3.5.1 節参照）．再帰データ型のパターンマッチが必要なときは，isinstance クラス合致判定関数でマッチを判断する．

```
class IndType(object):
 def __init__(self, argtuple=(), n=None):
  self.argtuple = argtuple + ((n,) if n else())
  self.next = n
```

※再帰的な定義をするコンストラクタのパラメタは n に格納し，それ以外のコンスタラクタのパラメタは argtuple に格納する．

図 8 再帰データ型のための Python3 用基底クラス
Fig. 8 A base class for inductive types in Python3.

```
class Difftype(IndType):
 def __init__(self, **k): super().__init__(**k)
class D_minus(Difftype):
 def __init__(self): super().__init__()
```

図 9 再帰データ型の変換例
Fig. 9 A translation example for inductive type.

```
from functools import partial
from inspect import getargspec
def isCanonicalizable(f):
    return len(getargspec(f).args) == 0
def curry(func):
    if not hasattr(func, '__call__'):
        return func
    if isCanonicalizable(func): return func()
    def paf(*args, **kwargs):
        f = partial(func, *args, **kwargs)
        if isCanonicalizable(f): return f()
        return lambda *a, **kwa:
                curry(f)(*a, **kwa)
    return paf
```

図 10 カリー化した関数を作成するメソッド 'curry'
Fig. 10 Method 'curry' to make curried functions.

```
leb = fix leb (n m : nat) {struct n} : bool :=
  match n with | 0 => true
               | S n' => match m with
                   | 0 => false
                   | S m' => leb n' m' end end
```

図 11 サンプル Coq コード
Fig. 11 A sample code in Coq.

また，関数に対するパラメタの部分的な適用は，Python3 の関数をカリー化した関数に変換（図 10）して対応した．

3.5.3 出力例

図 12 の Haskell コードが抽出される図 11 の Coq 実装の関数に対し，図 13 の Python3 コードが抽出される．

図 13 で applyArgsIfExist は，パラメタの部分適用のために用意した関数で，第 1 引数が空リストでなければその要素を引数として第 2 引数の関数に適用する．itse は，条件とそれが成立つときの処理の組を要素とするリストを受

```
leb :: Nat -> Nat -> Bool
leb n m = case n of {
  O -> True;
  S n' -> case m of {
    O -> False;
    S m' -> leb n' m'}}
```

図 12 抽出された Haskell コード
Fig. 12 The extracted code in Haskell.

```
def leb(n, m, *args):
    return applyArgsIfExist(
      args, itse(
        [(lambda: isinstance(n, O),
          lambda: True0())] +
        [(lambda: isinstance(n, S),
          lambda: itse(
            [(lambda: isinstance(m, O),
              lambda: False0())] +
            [(lambda: isinstance(m, S),
              lambda: applyPartially(
                leb, n.argtuple[0],
                m.argtuple[0]))],
            lambda: raise_(Exception(
              'Invalid ' + str(type(m))))))],
        lambda: raise_(Exception(
          'Invalid {}' + str(type(n))))))
```

図 13 抽出された Python3 コード
Fig. 13 The extracted code in Python3.

け取り，C 言語の switch 文の動作をする関数である．

3.6 非関数型言語へ変換した実装の性能向上の方策

変換後のコードの実行速度が遅くメモリを多く消費することがある．特に手続き型プログラム言語に変換する場合，変換元の MiniML が関数型プログラム言語寄りの言語仕様を持つため，メモリ，実行速度とも悪化が予想できる [*8]．

今回の対象プログラム（図 4）に関しては (1) 数値・リストを再帰データ型の nat, list で定義，(2) 再帰呼び出しを多用，の 2 点がメモリ量や速度の悪化の要因になりうる．

このうち（1）への対策は，python3 言語仕様の int 型，list 型を使う実装に変換する，よくある方法で行える [*9]．

次の（2）への直接の対策は，再帰呼び出しをループ実行に変更することだが，Coq の抽出機能のフレームワーク内での変更は容易ではない．ただ，再帰呼び出しのたびにリスト操作を行う方式ではなく，python3 の yield 式も用い generator を返す方式にすることで，メモリ量等を改善できると考える．

[*8] そのため Coq のプログラム抽出機構では個々の再帰データ型について，その定義とパターンマッチを行うコードに対する変換を，開発者が指示する仕組みが用意されている．
[*9] Extract inductive コマンドを扱えるよう変換ルーチンを追加．

3.7 実装

第 3.5 節で述べた Python3 変換モジュールを Coq バージョン 8.5 用に開発し，第 3.6 節の対応を実施中である．

下記の評価のために，第 3.6 節への対応完了後の変換モジュールから出力できるであろうコードも，（変換モジュールを使わず）手組みで作成した．ただし，再帰呼び出しのループ化は Coq の抽出機構への組み込みの目処が立っておらず，行っていない．

4. 評価

第 1 節で挙げた 3 点の RQ に関して，証明駆動開発が実用に耐えうるか確認する．確認は，第 3.5 節を実装したモジュールによって変換したコードに対して行う．また第 3.6 節の変換モジュール実装が完了した場合に変換できるであろうコード（以降「将来抽出可能なコード」とよぶ）でも確認する．

4.1 評価実験内容と結果

4.1.1 RQ1 に関して

Coq のプログラム抽出機構に組み込んだ Python3 変換器（以降，変換器とよぶ）で，図 4 のコードを Python3 コードに変換した．

変換中にエラーは報告されず，変換は正常終了した．

変換されたコードを，Python3 のコードチェッカである flake8 にかけ，問題点が指摘されないことを確認した．

4.1.2 RQ2 に関して

第 3.4 節で述べたとおり，今回の証明が自然数のリストに対する証明となっているため，抽出されたプログラムも自然数のリストを入力とする．そこで，実際に業務で使用しているデータの当該部分（これは文字列値のリスト 2 本である）を辞書順でソートし昇順に自然数を割り振って変換し，自然数のリスト 2 本を得た．このリスト 2 本を 1 組として，表 1 に示す 9 組を用意し，手組みのコード（図 1 に存在するバグを（図中のコメントを外して）修正したコードと抽出コードで実行結果を比較した．

比較の結果，いずれのリストでも同一の実行結果であることが確認できた．

4.1.3 RQ3 に関して

表 1 に示す 9 組の入力を Coq 記述から変換したコード（以降「Coq 由来コード」），将来抽出可能なコード，手組コード（修正後のコード）で処理し，パソコン（Core i7-5600U 2.6GHz，メモリ 8GB）上の VM で，実行完了までの所要時間とメモリ使用量を比較した．

CV[*10] を所要時間とメモリ使用量で比較すると前者 10%から 60%に対し後者 2%未満と前者で計測値のばらつきが大きかった．計測対象の処理による CPU 発熱で速度

[*10] Coefficient of variation．変動係数．複数回計測した値の標準偏差を平均値で割った値．小さいほどばらつきが小さい．

制限がかかった等の外部要因が影響していると考える．そのため，以降の比較ではメモリ使用量に関して論じる．

表 2 はメモリ使用量の比較である．各評価用データに対して 10 回計測し，その平均値を表の m_C（Coq 由来コード），m_H（将来抽出可能なコード），m_N （手組コード）に記した．

入力リストの要素数とメモリ使用量の関係のプロットと線形回帰モデルを図 14 に示す（左から Coq 由来コード，将来可能なコード，手組コードに対するメモリ使用量）．（注: 3 つのうち左端のグラフは他と X 軸の内容が異なる．）

Coq 記述から変換したコードは，手組コードに比べメモリ使用量の増加度合いが大きい．入力ひと組を構成するリスト 2 本それぞれの要素数を $|l_1|, |l_2|$ とすると，Coq 記述から変換したコードは $|l_1|^2+|l_2|^2$ に比例して大量のメモリを消費する（図 14 左）のに対し，手組のコードは $|l_1|+|l_2|$ に比例するメモリ消費に抑えられている（図 14 右）．

将来抽出可能なコードは，手組コードと同じく，$|l_1|+|l_2|$ に比例するメモリ消費に抑えられる（図 14 中）．ただし，係数は 10 倍異なり，手組コードの差し替えにはさらなる改善が必要である．

データ ID	リスト 1 の要素数	リスト 2 の要素数	データ ID	リスト 1 の要素数	リスト 2 の要素数
T1	69	74	M1	4152	3925
XS1	478	544	M2	3924	4147
XS2	431	500	L1	14479	16085
S1	1157	1714	L2	2598	33951
S2	1226	1784			

表 1 評価用データ（実適用先のデータから抽出）

Table 1 Data collected from practical data for evaluation.

データ ID	m_C (KB)	m_H (KB)	m_N (KB)	m_C/m_N	m_H/m_N
T1	11755	8018	7936	1.5	1.0
XS1	96768	8377	7967	12.1	1.1
XS2	83792	8358	7986	10.5	1.0
S1	711152	10122	8090	87.9	1.3
S2	772730	10145	8121	95.1	1.2
M1	（未計測）	14428	8429	N/A	1.7
M2	（未計測）	14445	8432	N/A	1.7
L1	（未計測）	40022	11014	N/A	3.6
L2	（未計測）	52292	12515	N/A	4.2

表 2 メモリ使用量

Table 2 Memory usage.

4.2 考察

4.2.1 実用性

RQ1, 2 に関しては実装に耐えうる評価結果をえた．

RQ3 に関して，空間計算量がそれぞれ $O(|l_1|^2+|l_2|^2)$, $O(|l_1|+|l_2|)$ と前者が悪いことに起因して，現状で Coq 由来コードは手組コードに比べ動作が遅い．

将来抽出可能なコードは，手組コードと似た比例関係であり，空間計算量が $O(|l_1|+|l_2|)$ 程度に改善する．この将来抽出可能なコードを，実際に抽出できる実装を用意して，Coq 由来コードが実用に耐える可能性がある．

4.2.2 バグ発見

今回の計測に先立って Coq 由来コードと手組コードの間で実行結果に差異がないことを確認した．その過程でデータ ID XS1 のデータに対して，両者のコードの実行結果が食い違うことがわかった．精査して，手組コード（図 1）にバグがあり図中のコメントアウト部分のような修正が必要と判明した．

この望外の出来事は形式的証明がテストより強力である（少なくともテストを補完できる）可能性を示している．

4.3 制限と妥当性への脅威

今回はアルゴリズム 1 事例に関して実験を行ったが，これが一般化できない可能性がある．また一般化できたとして，コード等価性の確認，できれば保証するための証明が必要である．ただし Coq のソースコードにはプログラム抽出のテスト用の入力コード [*11] があり，この全てに対して RQ1〜3 を確認すればこの懸念はある程度払拭できる．

将来抽出可能なコードは，メモリ使用量に関して手組コードと似た比例関係であるが，比例係数は 10 倍大きい（図 14 中，右）．また所要時間は正確に計測できていないが，Coq 由来コードに似た関係がみられた．これは，将来抽出可能なコードでの再帰呼び出しによる実装と，手組コードでのループ実装との違いの影響しており改善が必要と考える．アプローチとしてはループ実装への変更のほか，関数型プログラム言語で行われる末尾再帰に対する実装効率化を採り入れる方法がある．

最後に定理証明作業の作業負担増加が開発現場で受け入れられない可能性がある．現場試行で確認する必要がある．

5. 関連研究

5.1 言語 Ruby のプログラム抽出

手続き型言語 Ruby へのプログラム抽出がかなり早期に登場した [7]．関数型言語のみに現れる再帰データ型のパターンマッチ等の言語仕様を吸収するため，Ruby 記述のユーティリティ関数を用意してそれを呼び出すコードへ変換する方式を採っている．

[*11] Coq ソースコード中の test-suite/success/extraction.v など．

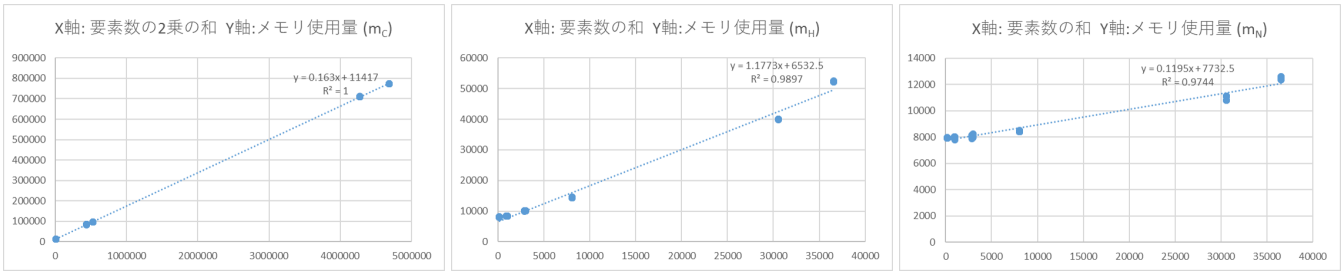

図 14 入力リストの要素数とメモリ使用量（Coq 記述，将来可能な記述，手組コード）の関係
Fig. 14 Relations b/w input size and memory usage by code from Coq (left), code extractable (mid.), and code directly written in native prog. language (right).

ただしユーティリティ関数で言語仕様差をすべて吸収する方式を採っているため，本稿第 3.5.1 節 で示したような，吸収しきれなかった場合の解決手段は提示されていない．また，関数型プログラム言語から手続き型プログラム言語を抽出したことによるメモリ使用量の悪化に対する方策は提案できていない．

一部の MiniML 仕様は未実装であり，残念ながら今回我々がプログラム抽出対象としたプログラム (図 4) は，その未実装部分を含んでいたため変換できなかった．

5.2 言語 Scala のプログラム抽出とアルゴリズム拡張

弱い型推論 (local type inference) しか行わない [*12] プログラム言語である Scala に変換するために，Coq の持つ抽出機能が実行するアルゴリズム $\mathcal{M}'$ が削り過ぎる型情報を残すアプローチ $\mathcal{M}'_{annot}$ [10] を作り出し，抽出に成功した [8]．その後，Coq の内部構造が変わったことにより方式の変更を迫られ，あらためて改良したアルゴリズム $\mathcal{M}''$ が作られ，さらに $\mathcal{M}''$ 自身の健全性，完全性が Coq で証明された [11]．

我々が抽出対象としたプログラム（図 4）も変換できる．

しかし，抽出後が関数型プログラム言語であるため，非関数型言語への対応方法や抽出後ノードの性能に関する言及はない．

6. おわりに

本稿では，実業務での適用を念頭に，形式的証明をプログラムの動作保証に活用するための方法を提案した．

実業務で稼働させるアルゴリズムに対して，その性質を Coq で形式的に証明し，証明済みのプログラムを，実業務で用いる非関数型プログラム言語に変換するモジュールを作成した．また関数型プログラム言語である Coq から非関数型プログラム言語に変換した際のメモリ使用量の悪化を防ぐ方策を提示し，10 倍程度の悪化にまで抑えられることを示した．

このための実験の過程で実プロジェクトのコード中に潜在バグを発見し，証明駆動開発が実際のテストより強力で有用である可能性を示した．

今後，上記提案のうち未実装の部分を完成させ，実際の開発に適用を試行していく予定である．

[*12] 例えば再帰呼び出しされる関数の帰り値の型を推論しないため，コード中に明示する必要がある．

参考文献

[1] 池渕未来：プログラミング Coq ～ 絶対にバグのないプログラムの書き方 ～，，入手先 ⟨http://www.iij-ii.co.jp/lab/techdoc/coqt/coqt8.html⟩（参照 2016-05-01）．
[2] Goodspeed, B.: Proof Driven Development, , available from ⟨http://arxiv.org/abs/1512.02102⟩ (accessed 2016-05-01).
[3] : The Coq Proof Assitant, , available from ⟨https://coq.inria.fr/⟩ (accessed 2016-05-01).
[4] : Isabelle, , available from ⟨https://isabelle.in.tum.de/⟩ (accessed 2016-05-01).
[5] Letouzey, P.: Certified functional programming Program extraction within Coq proof assistant (English version of his Ph.D. Dissertation), , available from ⟨https://www.researchgate.net/publication/280790704_Certified_functional_programming_Program_extraction_within_Coq_proof_assistant⟩ (accessed 2016-05-01).
[6] Filliâtre, J.-C. and Letouzey, P.: Extraction of programs in Objective Caml and Haskell, , available from ⟨https://coq.inria.fr/refman/Reference-Manual026.html⟩ (accessed 2016-05-01).
[7] Mizuho, H.: coq-ruby, , available from ⟨https://github.com/mzp/coq-ruby⟩ (accessed 2016-05-01).
[8] 今井宜洋，姜 帆：Coq2Scala, , available from ⟨http://proofcafe.org/wiki/Coq2Scala⟩ (accessed 2016-05-01).
[9] : ComCert - Compilers You Can Formally Trust, , available from ⟨http://compcert.inria.fr/compcert-C.html⟩ (accessed 2016-05-01).
[10] Imai, Y.: The Algorithm $\mathcal{M}'_{annot}$ for Coq Extraction to Statically Typed Languages without Type Inference (in Japanese), IT Planning, Inc (online), available from ⟨https://www.scribd.com/doc/94956541/mlnagoya⟩ (accessed 2016-05-01).
[11] 逸見 港，田辺良則，今井宜洋，萩谷昌己：Coq から scala へのコード抽出とその妥当性，研究集会「高信頼な理論と実装のための定理証明および定理証明器」(TPP 2014)，（オンライン），入手先 ⟨http://catalog.lib.kyushu-u.ac.jp/handle/2324/1520947/p081.pdf (実装: https://github.com/hemmi/coq2scala)⟩ (2014).