情報処理学会論文誌　プログラミング　Vol.10　No.1　14–28 (Jan. 2017)

定理証明器Coqの効率的な有限ドメイン関数ライブラリ

坂口 和彦[1,a)]　亀山 幸義[1,b)]

受付日 2016年5月10日, 採録日 2016年8月4日

概要：本研究は，対話的定理証明器 Coq の有限型と有限ドメイン関数に関する既存ライブラリを改良し，従来のライブラリを用いた証明をほとんど変更することなく，その証明から抽出されるプログラムの効率を大幅に改善したものである．Coq の SSReflect/Mathematical Components ライブラリは，有限型と有限ドメイン関数をサポートするライブラリ fintype と finfun を提供し，これらのデータに対する証明の手間を大幅に削減することに成功している．しかし，その証明からプログラム抽出の手法で作成したプログラムは，多くの場合において非常に効率が悪いという問題がある．本研究では，fintype や finfun を改善したライブラリを実装した．このライブラリを用いて作成した証明から抽出した OCaml コードは，既存ライブラリの場合と比べて非常に高速である．例として，行列積の計算では，計算時間をおよそ $\mathcal{O}(n^5)$ から $\mathcal{O}(n^3)$ へ改善できたことを示す．また，既存の SSReflect ライブラリを用いた Coq の証明は，ほとんど書き換えることなく本研究のライブラリを用いた証明となる．例として，Gonthier らの Feit-Thompson 定理の約 17 万行におよぶ形式証明が，わずか 10 行以下の修正で，本研究のライブラリを用いた証明にできたことを示す．

キーワード：対話的定理証明，Coq，SSReflect，有限型，有限ドメイン関数，プログラム抽出，証明のモジュール性

Efficient Finite-domain Function Library for the Coq Proof Assistant

Kazuhiko Sakaguchi[1,a)]　Yukiyoshi Kameyama[1,b)]

Received: May 10, 2016, Accepted: August 4, 2016

Abstract: We provide finite-domain function libraries in Coq, which improves the efficiency of code extracted from proofs without forcing one to rewrite the whole proofs which use existing libraries. The SSReflect/Mathematical Components of Coq provide the libraries to support finite types (fintype) and finite-domain functions (finfun), which allow one to drastically reduce the burden of writing proofs. While useful in proving, they have a serious problem in the performance of code extracted from proofs. In this study, we improve the fintype and finfun libraries, and show that OCaml code extracted from proofs using our libraries are much more efficient than those using the SSReflect libraries, and that existing proofs using the SSReflect libraries can be ported to the proofs using our libraries with very little modification. As concrete evidence for that, we provide a matrix-multiplication benchmark, whose time complexity has been improved from $\mathcal{O}(n^5)$ to $\mathcal{O}(n^3)$ by our libraries. We also demonstrate that the 170,000-line proof of Feit-Thompson theorem has been successfully ported where we only have to rewrite less than 10 lines of the whole proof.

Keywords: interactive theorem proving, Coq, SSReflect, finite type, finite-domain function, program extraction, proof modularity

1. はじめに

有限性（*finiteness*）は計算機科学において非常に重要な概念である．有限性に関連するものの中でも特に，定義域が有限集合であるような関数，すなわち**有限ドメイン関数**（*finite-domain function*）は，有限グラフ，有限オートマトン，行列，有限集合の冪集合など，様々なデータ構造の表現に有用である．また，有限集合 T から $\{0,\ldots,|T|-1\}$ への全単射を使うことで，多くのプログラミング言語で使われる**配列**と有限ドメインの関数 $f : T \to A$ を 1 対 1 に対応付け，計算機上で効率的に扱うことが可能である．対話的定理証明器 Coq [19] など構成的論理に基づく定理証明器は，

[1] 筑波大学大学院コンピュータサイエンス専攻
Department of Computer Science, University of Tsukuba, Tsukuba, Ibaraki 305–8573, Japan
[a)] sakaguchi@coins.tsukuba.ac.jp
[b)] kameyama@acm.org

証明から関数型プログラムを自動的に抽出する機能 [16] を持ち，正しさが保証されたプログラムが得られるという利点がある．本研究の目標は，有限ドメイン関数と配列型の間の対応付けと，Coq のプログラム抽出 [14] の機能を用いて，有限ドメイン関数に関する高速なプログラムを抽出するためのライブラリを作成し，効率と信頼性を両立させたプログラム作成手法の確立に貢献することである．

Coq 上に実装された有限ドメイン関数ライブラリとしては，SSReflect/Mathematical Components ライブラリ [13]（これ以降では単に SSReflect と書く）の有限型（fintype）と有限ドメイン関数（finfun）のライブラリがあげられる．この有限ドメイン関数は Coq に元々ある関数とは異なり，その関数が返す値を並べたリストを使って定義されている．これによって，有限ドメイン関数の相等関係は外延的なものとなる[*1]ため，等式に関する推論がやりやすくなるという利点がある．

SSReflect の fintype，finfun ライブラリは，証明記述においては非常に有用であるが，それらを使って記述した証明から抽出したプログラムは非常に非効率であるという問題がある．SSReflect の有限型は有限性の証拠としてその型の要素を漏れや重複なく並べたリストを要求するように定義されている．このような有限性の定義の下では，有限ドメイン関数の関数適用を計算するためにリストを先頭からたどって「引数がその有限型の何番目の値か」を調べなければならない．このため，プログラム抽出の方法だけをいかに工夫して有限ドメイン関数を配列型として抽出するようにしたとしても，インデックス指定での読み出しが定数時間でできるという配列の利点を生かしきれないという問題がある．そこで，本研究では以下の手順で有限ドメイン関数ライブラリから抽出されるプログラムを高速化した：

(1) 型 T が有限型（`finType`）であることを「ある自然数 n について，T から $\{0,\ldots,n-1\}$ への全単射が存在する」という形で再定義する．それに合わせて，要素の列挙（`enum`）や濃度の計算（`card`）などの基本的操作を再定義する（3，4 章）．
(2) 有限型の定義に使われる全単射を用いて，有限ドメイン関数の関数適用の計算を再定義する（5 章）．
(3) 有限ドメイン関数を配列型として抽出されるようにする．それに合わせて有限ドメイン関数を作る操作や関数適用が配列に関する効率の良い操作を組み合わせた形で抽出されるようにする（6 章）．

本研究の特長として，変更前の SSReflect ライブラリを我々が改良したライブラリに差し替えて証明を再度チェックする際に，非常に限定的な場合を除いて書き換えが不要であることがあげられる．このような性質は，（ライブラリの）モジュール性，互換性，（証明の）再利用性といった言葉で言い表せるが，本研究では主に**モジュール性**（*modularity*）という言葉を用いる．元々，SSReflect ライブラリはこのようなモジュール性を非常に重要視して作られている．それによって，SSReflect を用いて記述された証明は，Coq やライブラリのアップデートにともなって必要となる書き換えが非常に少なくなっている．

SSReflect のモジュール性を維持する様々な工夫の中で本研究に大きく影響しているものとして，ライブラリ内の重要な計算手続きの定義がロックされ，ある決められた方法でしか展開できないようになっていることがあげられる．この定義のロックによって証明を書く人に「定義を直接使わずに補題を使って証明する」ことを強制している[*2]．どうしても必要な場合にはこれらの定義を展開できるが，それは推奨されない証明方法であり，この証明方法はほとんど SSReflect ライブラリ内の基本的な補題の証明にしか使われていない．我々はこの仕組みを最大限に生かすことで，その推奨されない証明方法であるところの定義の展開を用いている箇所を除けば，SSReflect 向けに書かれた証明をそのまま使えるようにライブラリを開発した．なお，本研究でのライブラリの書き換えでは，ロックされた定義の変更だけではなく，有限性の特徴付けの変更などの大規模な改造を行っており，このような状況でモジュール性を維持するにはライブラリの変更を注意深く行う必要があった．

また，モジュール性が成り立つことを実証するための具体例として，Gonthier らの Feit-Thompson 定理（奇数位数定理）の形式証明 [12] を我々のライブラリに移植し，それにともない必要となる変更の量を調べた．Feit-Thompson 定理の形式証明は約 17 万行の Coq コードからなる非常に巨大なものだが，移植に必要な変更はわずか 10 行以下であった．モジュール性に関する本研究の貢献は，このようにライブラリの大規模な改造を行ったとしても，巨大な証明をほとんど変更なしに通せることを明らかにした点にある（8 章）．

本研究のライブラリの有用性を示すため，有限ドメイン関数を使ういくつかのプログラムに対して本手法を適用し，変更前の SSReflect ライブラリを用いた場合と性能の比較を行った[*3]．まず，簡単な例として行列積の計算を行うプログラムを抽出し，実行時間を計測した．その結果，本手法によって計算時間をおよそ $\mathcal{O}(n^5)$ から $\mathcal{O}(n^3)$ へ改善し，大幅に効率化されたことを確認した．また，より大きな例題として，プレスバーガー算術の決定可能性の証明にも本

[*1] Coq では一般に相等関係の外延性は成り立たない．

[*2] ただし，定義をロックすることの最も重要な目的は，証明における項の書き換えや定義の展開，型チェックにおける項の比較などの様々な場面で巨大な項が作られ，Coq 自体の動作が遅くなることを防ぐことにある．

[*3] なお，上述の手順 (3) で配列として抽出するようにしたことが原因で実行時間に大きな差が出てしまうと，本手法の最も重要なアイディアである T から $\{0,\ldots,n-1\}$ への全単射を使うようにしたことによる差分が分からないため，比較対象となる「変更前」のプログラムに対してもほぼ同様の変更をしている．

手法を適用し，その証明から抽出される決定手続きの高速化に利用できることを示した（7 章）．

1.1　本論文を読むうえでの注意点

- 本研究で用いた Coq，OCaml コード [17] を公開している．これは，改良した SSReflect ライブラリ，7 章の適用例，ベンチマークのための OCaml プログラムなどをすべて含んでいる．
- 引用したコードと我々が記述したコードを明確に区別し，一目見て判別できるようにするため，引用したコードには灰色の網掛けをしている．
- 一部のコードは分かりやすさを優先して元のコードから `Section`，`Variable` の除去，notation の展開，型注釈の付加・除去などの変更をしている場合がある．ただし，これらの変更によって本質的に意味が変化してしまうことはない．
- 本論文執筆時点での最新の Coq (8.5〜8.5pl2) にはプログラム抽出の実装にバグがあり，抽出されたプログラムが型検査に通らない場合がある．このバグは本研究の手法で抽出するプログラムにも影響するため，本論文の内容をすべて再現するには Coq 8.5beta3 を用いる必要がある．
- 本研究で用い，改良した SSReflect/Mathematical Components ライブラリのバージョンは 1.6 である．

2.　SSReflect の数学的構造

本研究における重要な道具として SSReflect の有限型があげられるが，SSReflect ライブラリでは他にも相等関係[*4]が決定可能な型（`eqType`），選択関数が構成できる型（`choiceType`），数え上げの可能な型（`countType`），さらに環（`ringType`）や体（`fieldType`）などの代数構造が定義されている．これらの構造はすべて，型と演算とその性質が組になったものであるという点で共通しており，すべて一貫した方法で定義されている．

SSReflect の有限型の定義や本研究での有限型の再定義の方法を述べるための導入として，本章では SSReflect での数学的構造の定義方法 [10] を簡単に説明する．

2.1　最も単純な例：eqType

まず，最も単純な例として等しさが決定可能な型 `eqType` の定義を説明する．上述の `choiceType`，`countType`，`finType` などは `eqType` から派生する形で定義されている．SSReflect で実際に使われている定義は説明に適さないため，以下ではそれを少し変形した，文献 [10] で使われている定義を引用する．

```
Module Equality.

Record mixin_of (T : Type) : Type :=
  Mixin {op : rel T;
         _ : forall x y, reflect (x = y) (op x y)}.
Structure type : Type :=
  Pack {sort :> Type; mixin : mixin_of sort}.
End Equality.

Notation eqType := Equality.type.
Notation EqType := Equality.Pack.
Definition eq_op T :=
  Equality.op (Equality.mixin T).
Notation "x == y" := (@eq_op _ x y).
```

`mixin_of` は，決定可能な二項関係 `op` と，それが相等関係であることの証明を要素として持つレコード型である．このレコードを **mixin** と呼ぶ．`type` は，型 `sort` と `sort` に関する mixin を要素として持つレコードである．すなわち型と決定可能な相等関係を組にしたものであり，「相等関係が決定可能な型」であると考えてよい．最後に定義されている `eq_op` は，`type` とその元を 2 つ引数にとって `op` で比較した結果を返す関数であり，`_ == _` は `eq_op` の 1 番目の引数を省略した表記法である．

`mixin_of`，`type`，そして次節で述べる `class_of` はすべての構造に共通する定義の形であり，それらとその構築子の名前もすべて共通している．SSReflect では各構造に対してモジュール（ここでは `Equality`）を用意することで，これらの名前の衝突を防いでいる．また，`Equality` モジュールの外では `type` とその構築子 `Pack` に対してそれぞれ `eqType`，`EqType` という別名を与えている．

`type` の定義には `sort :> Type` とあるため，`type` からその 1 番目の要素 `sort` への暗黙の型変換（coercion）が定義される．また，`type` は canonical structure [15] を用いて定義されているため，必要な場面で `type` の canonical なインスタンスを自動的に導出できるようになっている．たとえば，`nat` に関する `type` の（canonical な）インスタンス `nat_eqType` が定義されている状況で `2 == 3` すなわち `@eq_op _ 2 3` という式を書くと，`eq_op` の 1 番目の引数は自動的に `nat_eqType` で埋められ，型チェックが通るようになっている．さらに，これを応用して型 `T` の `eqType` インスタンスを（存在する場合には）`[eqType of T]` と書ける仕組みが用意されている．

2.2　派生をともなう場合

群にいくつかの公理を追加して環を作れるのと同様に，SSReflect で定義されている構造の多くは他の構造から派生する形で作られている．特に，本研究に関係する範囲では `eqType`，`choiceType`，`countType`，`finType` の順で派生関係にある [10] Fig. 3.

`eqType` から派生する `choiceType` の定義は，`eqType` の定義でそのベースとなる型 `sort` が `Type` となっていた部分を `eqType` に置き換え，以下のように書くのが自然であると考えられる．

[*4] ここでの相等関係は Coq の Leibniz equality, すなわち `Logic.eq` である．

```
Module Choice.
Record mixin_of (T : eqType) : Type :=
  Mixin { ... }.
Structure type : Type :=
  Pack {sort :> eqType; mixin : mixin_of sort}.
End Choice.
```

しかし，このように type を入れ子にする形で派生関係を表すと，coercion と canonical structure の仕組みが意図したとおりに働かない [10] §2.3, 2.4 ため，実際の choiceType の定義は以下のようになっている．

```
Module Choice.
Record mixin_of (T : Type) := Mixin { ... }.
Record class_of (T : Type) := Class
  {base :> Equality.class_of T; mixin : mixin_of T}.
Structure type :=
  Pack {sort :> Type; class : class_of sort}.
End Choice.
```

ここでの mixin_of は choiceType 特有の性質のみを記述した（eqType の性質は含まない）mixin であり，一方 class_of は eqType の mixin と新しく作った choiceType の mixin の組である．type は，型 sort と class_of sort の組である．上の 2 つの Choice モジュールの定義の差で重要な部分は，下の定義では type の定義に現れている sort が eqType ではなく Type となっており，eqType の公理は class に含まれているという点である．countType や finType の定義でも，同様の方法が用いられている．

3. 有限型

本章では，まず本研究での有限性の特徴付けの鍵となる ordinal 型について説明し，次に有限性の特徴付けをどのように変更したかを述べる．最後に，ライブラリ内で提供されている有限型のインスタンスをどのように再定義したか（一部については，なぜ再定義しなかったか）を述べる．

3.1 ordinal 型

本研究では，有限型を型 T と $\{0,\ldots,n-1\}$ の間の全単射を用いて再定義するため，$\{0,\ldots,n-1\}$ に相当する型が必要である．元々この型は有限型ライブラリ上で ordinal n 型（'I_n と表記する）として定義されている．本節では，有限性の特徴付けについて説明するための準備として ordinal 型について説明する．以下に ordinal 型の定義を示す[*5]．

```
Inductive ordinal (n : nat) : predArgType :=
  Ordinal m of m < n.
```

```
Notation "'I_' n" := (ordinal n) ... .
```

4.1 節で有限型の値の列挙 enum を再定義する際に，'I_n の要素を重複なく列挙したリストを構成する必要がある．これは，SSReflect で ord_enum 関数として定義されている．

```
Definition ord_enum (n : nat) : seq 'I_n :=
  pmap insub (iota 0 n).
```

iota 0 n は 0 から n - 1 までの自然数を順に並べたリストである．insub はここでは自然数から 'I_n への変換[*6]だが，返す値は option 'I_n であり，引数が n 以上で 'I_n の範囲を外れる場合には None を返す．pmap はリストに対して option 型の値を返す関数をマップし，None でない部分だけを集めたリストを返す関数である．

我々は，この ord_enum を以下のように再定義して抽出プログラムの効率を改善した．

```
Fixpoint ord_enum_rec n m :
  m <= n -> seq 'I_n -> seq 'I_n :=
  match m with
  | 0 => fun _ xs => xs
  | m'.+1 => fun (H : m' < n) xs =>
    @ord_enum_rec n m' (ltnW H) (Ordinal H :: xs)
  end.
Definition ord_enum n : seq 'I_n :=
  ord_enum_rec (leqnn n) [::].
```

ord_enum_rec は，証明の部分を無視すれば，自然数 m と自然数のリスト xs をとり，[0, ..., m - 1] ++ xs を返す関数である．この関数は m について再帰的に定義されている．m = 0 の場合には xs を返し，m = m'.+1 の場合には ord_enum_rec m' (m' :: xs) を返す．ただし，実際には 'I_n のリストを返すため，H : m <= n を引数にとっている．再帰呼び出しでは m' <= n の証明が必要だが，H : m' < n からそれを導く補題が ltnW である．

ord_enum n を再定義するには，ord_enum_rec の引数に n, n, 空のリストを渡せばよい．また，その際に n <= n を示す必要があるが，この証明は leqnn n と書ける．

この ord_enum の定義は元の定義と比較すると，n 以下であることの証明を持ち回すことで n との比較を不要にしており，iota 0 n 相当の nat のリストを作る必要もない．なお，証明項はプログラム抽出によって除去されるので効率に影響しないことに注意されたい．

3.2 有限性の特徴付け

本節では，本研究で有限型の mixin をどのように再定義したかを述べる．有限型は Finite モジュール内で定義されている．まず，元の SSReflect ライブラリでの定義を以下に示す．

```
Definition axiom (T : eqType) e :=
  forall x : T, count_mem x e = 1.

Record mixin_of (T : eqType) := Mixin {
  mixin_base : Countable.mixin_of T;
  mixin_enum : seq T;
  _ : axiom mixin_enum
}.
```

[*5] predArgType については 4.1 節で説明するが，ここでは Type と同義だと考えてかまわない．

[*6] 実際には，ある型に対して部分集合となっているような型（subType）すべてに対して使える．

axiom の定義で使われている count_mem x e は，リスト e 中の x の出現回数を表す自然数である．よって，axiom mixin_enum は「型 T の任意の値 x について，mixin_enum 中の x の出現回数はちょうど 1 回」すなわち「mixin_enum は型 T の値を漏れや重複なく並べたリストである」ことを意味している．Countable は countType を定義するモジュールである．Finite.mixin_of T 型の値は T が有限型であることの証拠となっている．

次に，変更後の定義を以下に示す．

```
Record mixin_of (T : eqType) := Mixin {
  mixin_base   : Countable.mixin_of T;
  mixin_card   : nat;
  mixin_encode : T -> 'I_mixin_card;
  mixin_decode : 'I_mixin_card -> T;
  _ : cancel mixin_encode mixin_decode;
  _ : cancel mixin_decode mixin_encode;
}.
```

mixin_card は，型 T の濃度である．このとき，T から 'I_mixin_card への全単射を与えることで，T の有限性が示せる．T から 'I_mixin_card への全単射は，T から 'I_mixin_card への関数 mixin_encode とその逆関数 mixin_decode で与えられている．本研究では，これらの関数をそれぞれ**符号化関数**（*encoding function*），**復号化関数**（*decoding function*）と呼ぶ．これが全単射であることは，これらの関数をどちら向きに合成しても恒等関数になる（cancel）ことによって保証している．

3.3　mixin の構築子

前節で示した mixin の構築子 Finite.Mixin は T : eqType と T の countType の mixin を別々に引数にとるため，有限型のインスタンスを作る際には別途定義した countType を引数にとる mixin の構築子を用いる．また，そのような種類の構築子は 1 つではなく，それによって何通りかの有限性の証拠の与え方に対応している．以下に，その例を示す．

```
FinMixin (EnumMixin) :  forall T (e :  seq T),
    Finite.axiom e -> Finite.mixin_of T
UniqFinMixin :  forall T (e :  seq T),
    uniq e -> e =i T -> Finite.mixin_of T
```

FinMixin は，Finite.axiom を有限性の証拠として用いる mixin の構築子である．EnumMixin は FinMixin の Finite モジュール内での名前であり，外から見える名前は FinMixin である．一方，UniqFinMixin は Finite.axiom e と同値な別の条件を用いる mixin の構築子である．その条件は，e に重複がなく（uniq e），かつ漏れがない（e =i T[*7]）ことである．ここで重要なのは，FinMixin 以外の構築子もすべて FinMixin を用いて定義されているという点である．

[*7] x =i y は，x と y が述語として外延的に同等であることを示す表記法である．右辺の型 T は，ここでは T の要素すべてに対して真であるような述語を意味する．

我々は，前節で ordinal 型への全単射を用いる形で有限性を再定義した．一方，上で示したように SSReflect の有限型ライブラリでは様々な種類の有限性の証拠から有限型を構成できるようになっており，これらを新しい有限性の定義の下で再現しなければ，元々定義されていた多くの有限型が使えなくなる．SSReflect の有限型ライブラリで提供されているすべての mixin は，EnumMixin を用いて定義されているので，ここでは我々が採用した有限性の定義の下で EnumMixin を再現する方法を説明する．EnumMixin は Finite.axiom を満たすリスト e を有限性の証拠として要求するものであったので，そのような e から T と 'I_(size e) の間の全単射を構成する．以下にその符号化・復号化関数の型を示す[*8]：

```
EnumMixin_enc :  axiom e -> T -> 'I_(size e)
EnumMixin_dec :  'I_(size e) -> T
EnumMixin_encodeK :  forall (H : axiom e),
    cancel (EnumMixin_enc H) EnumMixin_dec
EnumMixin_decodeK :  forall (H : axiom e),
    cancel EnumMixin_dec (EnumMixin_enc H)
```

この全単射を構成するにあたって注意すべき点として，e が axiom を満たすことを仮定しないと，符号化関数になりうるような T から 'I_(size e) への関数が構成できないということがあげられる．よって，EnumMixin_enc は引数に axiom e をとるように定義されている．

次に，新しい有限性の定義から自然に導かれる mixin の構築子 BijOrdMixin を，EnumMixin の定義を模倣する形で定義する．

```
BijOrdMixin :
  forall T (n : nat) (f : T -> 'I_n)
    (g : 'I_n -> T),
  cancel f g -> cancel g f -> mixin_of T
```

最後に，上で構成した全単射と BijOrdMixin を組み合わせることで，EnumMixin を再現できる．

```
Definition EnumMixin T (e : seq T)
    (H : @axiom T e) :
  mixin_of T :=
  BijOrdMixin (@EnumMixin_encodeK T e H)
              (@EnumMixin_decodeK T e H).
```

これによって元々 SSReflect にあった有限型のインスタンスはすべて扱えるようになった．ただし，本研究の恩恵を受けるには可能な限り多くの有限型のインスタンスを BijOrdMixin を用いて再定義する必要がある．次節では，個々の有限型のインスタンスの再定義について説明する．

3.4　有限型のインスタンス

本節では，いくつかの重要な有限型を例にとり，本研究

[*8] ここでは先頭の forall (T : eqType) (e : seq T), を省略している．

でそれらの有限型のインスタンスをどのように再定義したかを述べる．まず，最も簡単な ordinal の有限型の定義を見る．以下が変更前の定義である．

```
Lemma ord_enum_uniq n : uniq (ord_enum n).
Lemma mem_ord_enum n i : i \in (ord_enum n).
Definition ordinal_finMixin n :=
  Eval hnf in
  UniqFinMixin (ord_enum_uniq n) (mem_ord_enum n).
Canonical ordinal_finType n :=
  Eval hnf in FinType 'I_n (ordinal_finMixin n).
```

ここでは，'I_n の有限性の証拠として 3.1 節で見た ord_enum を使っている．UniqFinMixin を用いて 'I_n の有限型の mixin を構成するには，uniq (ord_enum n) と ord_enum n =i 'I_n を示せばよい．最後に，'I_n の有限型インスタンス ordinal_finType n を定義している．FinType は 2.1 節で見た EqType と同様に，有限型の Pack の別名である．

次に，変更後の ordinal_finMixin の定義を以下に示す．なお，ordinal_finType の定義は変更していない．

```
Lemma cancel_id T : @cancel T T id id.
Definition ordinal_finMixin n :=
  Eval hnf in
  BijOrdMixin (@cancel_id 'I_n) (@cancel_id 'I_n).
```

BijOrdMixin を用いて 'I_n の有限型の mixin を構成するには，'I_n と 'I_n の間の全単射が必要である．このような全単射は明らかに恒等関数で十分であるため，恒等関数どうしを合成すると恒等関数になること（cancel_id）を証明し，それを使って ordinal_finMixin を定義している．

BijOrdMixin を用いて再定義した有限型のインスタンスは，他にも unit，bool，有限型の option 型，有限型と有限型の直和（sum）や直積（prod），有限型から有限型への（有限ドメイン）関数などがある．特に，直和，直積，関数は符号化・復号化関数を簡単な算術のみで構成できる興味深い例である．一方で，有限型の部分型，有限型と有限型の依存和（tag_finType）などに対しては単純な数式で符号化・復号化関数を構成できなかったため，FinMixin を用いる定義のままになっている．

以下に，直和 $A \sqcup B$ と直積 $A \times B$ の濃度，符号化・復号化関数の定義を示す．関数の有限型インスタンスについては 5.2 節を参照されたい．

$$|A \sqcup B| = |A| + |B|$$

$$\mathrm{enc}_{A\sqcup B}(x) = \begin{cases} \mathrm{enc}_A(y) & (x = \mathsf{inl}(y)) \\ |A| + \mathrm{enc}_B(z) & (x = \mathsf{inr}(z)) \end{cases}$$

$$\mathrm{dec}_{A\sqcup B}(i) = \begin{cases} \mathsf{inl}(\mathrm{dec}_A(i)) & (i < |A|) \\ \mathsf{inr}(\mathrm{dec}_B(i - |A|)) & (|A| \le i) \end{cases}$$

$$|A \times B| = |A| \times |B|$$

$$\mathrm{enc}_{A\times B}(x, y) = \mathrm{enc}_B(y) + \mathrm{enc}_A(x) \times |B|$$

$$\mathrm{dec}_{A\times B}(i) = \left(\mathrm{dec}_A\left(\left\lfloor \frac{i}{|B|} \right\rfloor\right), \mathrm{dec}_B(i \bmod |B|)\right)$$

4. 有限型に関する基本的計算

有限性を利用すると，様々なことが計算できる．たとえば，

(1) 有限型に属する値を 1 つ 1 つ列挙する，
(2) 有限型の濃度（自然数）を計算する，
(3) 適当な自然数 n について，有限型と $\{0, \ldots, n-1\}$ の間の全単射を構成する，

といったことができる．SSReflect の有限型はこのような有限性を生かした計算手続きを提供している．本章では，それらの計算手続きがどのようにして定義されており，本研究でどのように再定義したかを述べる．

また，4.1 節では，SSReflect ライブラリの中で広く使われている定義の展開を抑制する（ロックする）方法について説明する．これは，本研究の特長としてあげているモジュール性を成り立たせるための重要な要素である．

4.1 enum：値の列挙

SSReflect の有限型ライブラリにおいて最も基本的な計算手続きは，有限型に属する値の列挙 enum である．本節のこれ以降の内容で示す有限型に関する計算手続きは，元のライブラリではすべて enum を使って定義されている．

SSReflect では，意図しない場所で定義が展開されることを防ぐため，多くの計算手続きの定義は simpl タクティクで展開されないように定義されているか，もしくはより厳格に unfold や compute などでも展開できないように定義されている．enum の定義も例外ではなく，後者の方法で定義の展開が制限されている．以下に Finite モジュール内の enum の定義を示す．

```
Module Type EnumSig.
Parameter enum : forall cT : type, seq cT.
Axiom enumDef :
  enum = fun cT => mixin_enum (class cT).
End EnumSig.

Module EnumDef : EnumSig.
Definition enum cT := mixin_enum (class cT).
Definition enumDef := erefl enum.
End EnumDef.
```

EnumDef.enum は有限型に属する値を列挙する関数であり，これは単に有限型の mixin からリスト mixin_enum を取り出すことによって定義されている．EnumDef.enum の定義にモジュールを用いているのは，いかなる方法でも定義を展開できないようにするためである．Coq では Module def : sig. でモジュールの定義を開始すると，そのモジュールの中身は展開できなくなる[*9]．しか

[*9] シグネチャを書く必要があり，かつモジュールの中身を展開可能にするには，: の代わりに <: と書く．

し，これだけでは定義がまったく展開できず証明もできないので，EnumDef.enumDef によって EnumDef.enum = fun cT => mixin_enum (class cT) [*10]を保証している．EnumDef.enumDef を使って証明する箇所を最低限に抑えることで，定義を変更したときに必要な証明の修正が限定的になっている．

我々は，EnumDef.enum を以下の形で再定義した．この定義で使われている raw_card T は T の濃度である．

```
Definition raw_card T := mixin_card (class T).
```

```
Definition enum cT :=
  map (mixin_decode (class cT))
      (ord_enum (raw_card cT)).
```

SSReflect には EnumDef.enum のような形でロックされた定義を「アンロック」するための枠組みが備わっている．以下のように記述することで，rewrite unlock と書くだけで EnumDef.enum の定義が展開されるようになる．

```
Canonical finEnum_unlock :=
  Unlockable Finite.EnumDef.enumDef.
```

さらに，EnumDef.enum を用いて，有限型の部分集合に対する値の列挙 enum が以下のように別途定義されている．これは Finite モジュールの外で定義されており，ユーザが直接使う enum はこちらである．

```
Definition enum_mem T (mA : mem_pred T) :=
  filter mA (Finite.EnumDef.enum T).
Notation enum A := (enum_mem (mem A)).
```

enum は引数に有限型の部分集合をとるが，それは [pred x : 'I_10 | odd x] のような述語だけでなく，'I_10 のような有限型や [:: false] のような有限型の値からなるリストも含まれる．有限型を引数にとる場合には全体集合として解釈され，リストはそのリストに含まれる値の集合として解釈される．このように，様々な値を述語として扱う仕組みを提供するのが mem_pred と mem である．

mem は forall T (pT : predType T), pT -> mem_pred T という型を持ち，predType T に属する型と mem_pred T はどちらも T 上の決定可能な述語を意味する．mem に渡せる「部分集合」としては，predArgType に属する型[*11]，eqType に属する型の値のリスト，finset ライブラリで定義される冪集合型の値などがある．

4.2　card：濃度の計算

次に，有限型の部分集合 A の濃度 card A（#|A|と表記する）の定義について説明する．部分集合 A の濃度は，enum A の長さで定義できる．

```
Definition card (T : finType) (mA : mem_pred) :=
  size (enum_mem mA).
```

card の定義も Finite.enum と同様に，モジュールを用いてロックされている．card に対して表記法 #|A| を与える．

```
Notation "#| A |" := (card (mem A)) ... .
```

上述のように有限型の部分集合に対して定義された濃度は一般的で扱いやすい．しかし，我々は有限型を先に濃度を与える形で定義しているため，（部分集合ではない）有限型の濃度を計算したい場合にはこちらを直接使った方が計算コストが小さく済む．前節ですでに raw_card という名前で有限型の濃度を定義しているため，これに $|T| という表記法を与える[*12]．また，有限型 T について #|T| と $|T| が同等であることを証明した．

```
Notation "$| T |" := (Finite.raw_card T) ... .
```

```
Lemma cardT' T : #|T| = $|T|.
```

上に示した card の定義は，一度部分集合に属する値のリストを作ってからその長さを計算しており，非効率である．そこで ord_enum と同様にして，card をリストを介さずに計算するように再定義した．

```
Fixpoint
  card_def_rec (T : finType) (mA : mem_pred T) n :
  n <= $|T| -> nat :=
  match n with
  | 0 => fun _ => 0
  | n'.+1 =>
    fun (H : n' < $|T|) =>
      mA (Finite.mixin_decode
            (Finite.class T) (Ordinal H)) +
      card_def_rec mA (n := n') (ltnW H)
  end.
Definition card (T : finType) (mA : mem_pred) :=
  @card_def_rec T mA $|T| (leqnn $|T|).
```

4.3　符号化・復号化

本研究では，有限性の特徴付けとして ordinal 型との間の全単射（符号化・復号化関数）を採用している．SSReflect には元々これらに相当する関数が定義されている[*13]:

enum_rank : forall T, T -> 'I_#|T|

enum_val : forall T (A : pred T), 'I_#|A| -> T

enum_rank が符号化関数，enum_val が復号化関数である．前者は有限型について定義されており，後者は有限型の部分集合について定義されている．有限型の部分集合に関する符号化関数 enum_rank_in も存在するが，ここでは説明しない．

我々はこの enum_rank，enum_val とは別に，有限性の特

[*10] erefl enum は enum = enum の証明だが，EnumDef モジュール内では enum の定義が展開できるため，この等式の証明となる．
[*11] ordinal 型は元々 predArgType に属するものとして定義されているため，mem の引数としてとれる（3.1 節）．
[*12] raw_card の定義を直接 Notation の中身として書いてしまうと，pretty-print 時にこの表記法が有効にならないという問題があったため，定義を 1 つ挟むことで解決した．
[*13] ただし，集合の要素を列挙したリストを使って計算しているため，先に全単射を与えたのとは異なり，非常に計算が遅くなる．

徴付けに符号化・復号化関数を用いていた．それらの関数は適切に実装されていれば enum_rank，enum_val より高速であり，有限ドメイン関数の実装に適している．しかし，これらの関数はそのままではロックされてないため定義が展開可能であり，その型は mixin の中に埋め込まれた有限型の濃度を使って書かれているので扱いにくい．そこで，

(1) enum や card と同様の方法で定義をロックし，
(2) T -> 'I_#|T| や 'I_#|T| -> T のように扱いやすい型で再定義する，

必要がある．定義のロックについては enum や card と同様であるため省略する．型を 'I_#|T| に合わせるには ordinal 型を自然数の等式を使ってキャストする関数 cast_ord を用いて以下のよう記述する．

```
Notation raw_fin_encode :=
  (Finite.mixin_encode (Finite.class _)).
Notation raw_fin_decode :=
  (Finite.mixin_decode (Finite.class _)).
```

```
Definition
  fin_encode (T : finType) (x : T) : 'I_#|T| :=
  cast_ord (esym (cardT' T)) (raw_fin_encode x).
Definition
  fin_decode (T : finType) (i : 'I_#|T|) : T :=
  raw_fin_decode (cast_ord (cardT' T) i).
```

これらの符号化・復号化関数について，どちら向きに合成しても恒等写像になる（全単射である）ことと，enum T を ord_enum と fin_decode で書き直せることを証明した．

```
Lemma fin_encodeK T :
  cancel (@fin_encode T) (@fin_decode T).
Lemma fin_decodeK T :
  cancel (@fin_decode T) (@fin_encode T).
Lemma enumT' (T : finType) :
  enum T = [seq fin_decode i | i <- ord_enum #|T|].
```

5. 有限ドメイン関数

本章では，SSReflect の有限ドメイン関数とその再定義について説明する．有限型 T から型 A への有限ドメイン関数の実体は，型 A の値を #|T| 個並べた組（タプル）である．本章の前半では SSReflect のタプルライブラリについて説明する．本章の後半では，有限ドメイン関数ライブラリについて説明する．

5.1 タプル

型 A の値の n 個組は n.-tuple A という型で表される．タプル型は以下のようにリストとその長さに関する証明の組で定義されている．

```
Structure tuple_of (n : nat) (T : Type) : Type :=
  Tuple {tval :> seq T; _ : size tval == n}.
```

タプルを用いて有限ドメイン関数を作るには，

(1) タプルの要素をインデックス指定で取り出す操作（関数適用に必要）
(2) 定義域が有限の関数 f : T -> T' が返す値を並べたタプルを作る操作（関数からタプルへの変換に必要）

が必要となる．これらは，タプルのライブラリでそれぞれ tnth，codom_tuple 関数として定義されている．

```
tnth : forall n T, n.-tuple T -> 'I_n -> T
codom_tuple :
    forall T T', (T -> T') -> #|T|.-tuple T'
```

5.2 タプルの有限型インスタンス

有限型 T について，タプル n.-tuple T もまた有限型である．我々は，この有限型のインスタンスを ordinal との間の全単射を用いる形で再定義した．以下に有限型のタプルの濃度，符号化，復号化関数の定義を示す．

$$|T^n| = |T|^n$$

$$\mathrm{enc}_{T^n}([x_0, \ldots, x_{n-1}]) = \sum_{k=0}^{n-1} \mathrm{enc}_T(x_k)|T|^k$$

$$\mathrm{dec}_{T^n}(m) = \left(\mathrm{dec}_T\left(\left\lfloor \frac{m}{|T|^i} \right\rfloor \bmod |T|\right)\right)_{i<n}$$

この符号化・復号化関数は，そのまま有限ドメイン関数の有限型インスタンスの定義に使われるという点で重要である．また，これらの Coq 上での定義と全単射であることの証明は，いくつかの理由で複雑なものとなった：

(1) enc_{T^n} の右辺と dec_{T^n} の右辺の dec_T の引数は ordinal 型であるため，それぞれ $|T|^n$，$|T|$ より小さいことの証明が必要である．
(2) SSReflect で enc_{T^n} の右辺にあるような有限個の値の総和を書く場合，通常は bigop ライブラリ [1] を用いる．しかし，bigop ライブラリは tuple ライブラリに依存しており，この符号化関数の定義で bigop ライブラリを用いることはできない．
bigop の代わりに自然数のリストの総和を求める関数 sumn を用いることになったが，sumn は bigop と比較すると補題が不足しており，証明の手間が増えた．

5.3 有限ドメイン関数

有限型 aT から型 rT の有限ドメイン関数は，rT の要素からなる長さ #|aT| のタプルを用いて定義されている．

```
Inductive finfun_type
    (aT : finType) (rT : Type) : predArgType :=
  Finfun of #|aT|.-tuple rT.
```

SSReflect では，T から A への有限ドメイン関数を {ffun T -> A} と書く．特に定義域が 'I_n の場合，A ^ n とも書く．

通常の関数と有限ドメイン関数を相互に変換する関数の定義を以下で見る．有限ドメイン関数から通常の関数への変換 fun_of_fin は，有限ドメイン関数についての関数適用ができればよいので，tnth と enum_rank（符号化関数）

を使って定義する．通常の関数から有限ドメイン関数への変換 finfun は，codom_tuple がそのまま使える．これらの定義も，enum や card と同様の方法でロックされている．

```
Definition fgraph (aT : finType) (rT : Type) f :=
  let: Finfun t := f in t.
```

```
Definition fun_of_fin aT rT f x :=
  tnth (@fgraph aT rT f) (enum_rank x).
Definition finfun aT rT f :=
  @Finfun aT rT (codom_tuple f).
```

我々は，fun_of_fin の定義に使われている enum_rank を，fin_encode で置き換えた．enum_rank は数え上げによってインデックスを計算する一方，fin_encode は 3.4 節にあるように多くの有限型に対して効率的にインデックスを計算する．これによって有限ドメイン関数の関数適用が高速になる．

```
Definition fun_of_fin aT rT f x :=
  tnth (@fgraph aT rT f) (fin_encode x).
```

6. プログラム抽出

Coq は構成的論理に基づいており，証明をプログラムと見なすことができる．Coq のプログラム抽出機能は，非常に大きな項となりがちな証明から，プログラムとして不要な部分を除去するとともに，高階論理の項である証明を OCaml や Haskell などのプログラムに変換するものである．本研究は，有限型や有限ドメイン関数のライブラリを変更したうえで，Coq のプログラム抽出機能はそのまま用い，抽出プログラムの効率の改善を行った．

本章では，Coq のプログラム抽出機能を利用するにあたり，有限型・有限ドメイン関数に関する Coq の型や項を，抽出先言語のどのような型や項に対応させたかを説明する．Coq のプログラム抽出では，このような定義の読み替えを指定できる対象は 2 種類ある．

(1) Inductive や Record で定義された帰納的データ型[*14]（Extract Inductive で指定）
(2) Definition などで書いた計算手続きの定義（Extract Constant で指定）

4.3 節で定義した fin_encode，fin_decode は，型を合わせるために cast_ord を用いて定義されている．この cast_ord の引数として現れる #|T| は計算にはいっさい使われず，かつ T の濃度に対して線形に時間がかかるため，cast_ord を除去した形で抽出されるようにする．

```
Extract Constant EncDecDef.fin_encode =>
  "(fun t x -> (Finite.coq_class t)
               .Finite.mixin.Finite
               .mixin_encode x)".
```

[*14] 一方，Definition で「別名」を付けたデータ型に対して抽出先のプログラムでの型を指定する方法はない．よって，単に別名を付けるだけであっても，Inductive を使っておくとプログラム抽出には便利な場合がある．

```
Extract Constant EncDecDef.fin_decode =>
  "(fun t i -> (Finite.coq_class t)
               .Finite.mixin.Finite
               .mixin_decode i)".
```

次に，タプル型を配列にする．帰納的データ型を抽出先の別のデータ型に対応させるには，各コンストラクタとパターンマッチ（デストラクタ）に対応する関数が必要となる．タプル型の中身はリストなので，配列とリストの間の相互の変換がこれに対応する．よって，OCaml の Array.of_list がコンストラクタ，Array.to_list がデストラクタとなる．

```
Extract Inductive
  tuple_of => "array"
                ["Array.of_list"]
                "(fun f t -> f (Array.to_list t))".
```

これだけではタプルに関する計算をするたびに上で示したコンストラクタとデストラクタが呼ばれ，リストを介して計算するため非効率である．そこで，抽出されたコードの中の Array.of_list と Array.to_list が含まれる箇所を探し，それらを適切な OCaml プログラムに対応させた．まず，tnth と codom_tuple の抽出先コードを以下のように定義した．t.(i) は配列 t の i 番目，Array.init はサイズとインデックスから配列の要素を計算する関数をとり，配列を作る関数である．

```
Extract Constant tnth => "(fun _ t i -> t.(i))".
```

```
Extract Constant codom_tuple =>
  "(fun t f ->
     Array.init
     t.Finite.mixin.Finite.mixin_card
     (fun i -> f (EncDecDef.fin_decode t i)))".
```

5.2 節で示した有限ドメイン関数の有限型インスタンスの定義に使われる符号化・復号化関数は，そのままでは非効率な部分があるため，プログラム抽出の段階でその問題を解消した．まず，符号化関数は総和のインデックスの動く範囲のリストを一度作って畳み込みを行う定義になっていたため，そのようなリストを介さずに計算するように再定義した．

```
Extract Constant FinTuple.fin_encode =>
  "(fun n t x ->
    let rec loop i acc =
      if i = 0
        then acc
        else loop
             (i - 1)
             (acc * t.Finite.mixin.Finite.mixin_card
              + EncDecDef.fin_encode t x.(i))
    in loop n 0)".
```

復号化関数は Array.of_list を使うコードが抽出されていたため，OCaml の Array.init を用いて書き直した．

```
Extract Constant FinTuple.fin_decode =>
  "(fun n t i ->
```

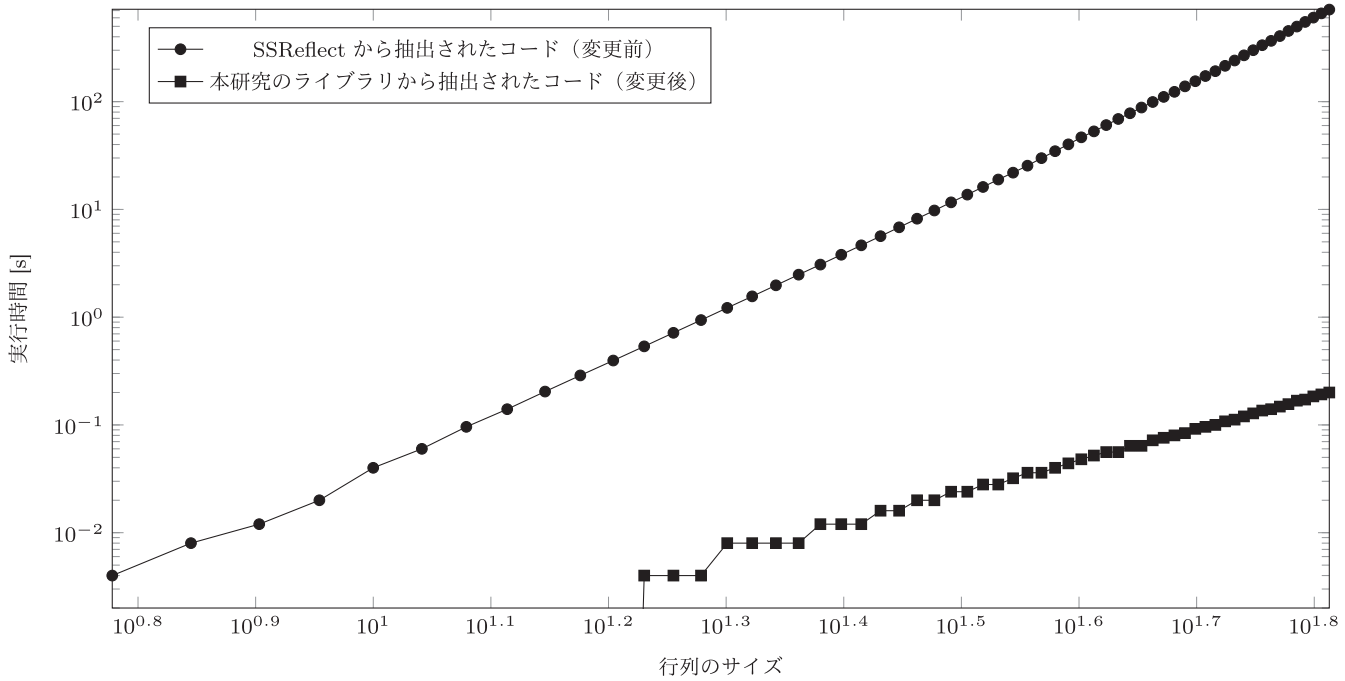

図 1　行列積の計算の実行時間
Fig. 1　Execution time of matrix multiplication.

```
    Array.init n
    (fun j -> EncDecDef.fin_decode t
      ((i / expn t.Finite.mixin.Finite.mixin_card j)
          mod t.Finite.mixin.Finite.mixin_card)))".
```

有限ドメイン関数から関数への変換（関数適用）にも，`EncDecDef.fin_encode`，`EncDecDef.fin_decode` と同様に計算に使われない `#|aT|` が出現していたため，プログラム抽出の段階で除去した．

```
Extract Constant FunFinfun.fun_of_fin =>
  "(fun aT f x -> f.(EncDecDef.fin_encode aT x))".
```

上記と同様の最適化の一部は，SSReflect における有限ドメイン関数の定義のままでも可能である．すなわち，タプルを配列に読み替え，`tnth` の最適化を行うことはできる．しかし，`codom_tuple` や `FunFinfun.fun_of_fin` の定義は符号化・復号化関数を使っているため，同様の最適化を変更前の SSReflect ライブラリに対して行うことはできない．

7. 適用例と性能評価

本章では，有限ドメイン関数を用いるプログラムの例を 2 つ示す．また，それらのプログラムに対して本研究の手法を適用することで，

(1) どの程度プログラムや証明の修正が必要であったか，
(2) プログラムのどの部分に対して本手法が作用し，実行時間が短縮するか，
(3) どの程度実行時間を短縮できるか，

を説明する．なお，性能評価に用いた計算機は MacBook Pro（Intel Core i5-4288U 2.60 GHz，16 GB RAM），OS は Debian GNU/Linux（amd64，unstable），OCaml コンパイラは OCaml 4.02.3 の ocamlopt である．実行時間の計測には OCaml の `Sys.time` 関数を用いている．本章で示すプログラムの実行時間は，すべて 5 回実行して計測したうちの中央値を採用している．

7.1 行列の積

本適用例では，SSReflect の行列・線形代数ライブラリ [11] を用いて行列積の計算を行った．SSReflect での $m \times n$ 行列は，`'I_m * 'I_n` を定義域とする有限ドメイン関数を使って定義されている．本研究での計算の高速化はこの部分に対して作用し，それによって行列の作成，行列の値の取り出し，行列積の計算などを高速化する．以下に行列型の定義を示す．

```
Inductive matrix
  (R : Type) (m n : nat) : predArgType :=
  Matrix of {ffun 'I_m * 'I_n -> R}.
```

整数を要素とする $n \times n$ 行列 $A = (i+j)_{i,j}$ に対して，行列の積 $A \cdot A$ を計算し，その計算時間を計測した．この行列積の計算に用いた Coq コードを以下に示す．この適用例のために新たに書いた Coq コードはこの `matrix_mult_test` だけであり，ライブラリの差し替えにともなう変更は不要であった．

```
Definition matrix_mult_test (n : nat) :=
  let mx := (\matrix_(i < n, j < n) (i%:Z + j%:Z))%R
  in (mx *m mx)%R.
```

サイズ n（$6 \leq n \leq 65$）の行列に対する実験結果のグラフを**図 1** に示す．この実験結果から，変更前の計算時間は $\mathcal{O}(n^5)$ で近似でき，変更後の計算時間は $\mathcal{O}(n^3)$ で近似でき

表 1　プレスバーガー算術の決定手続きの実行時間
Table 1　Execution time of the decision procedures for Presburger arithmetic.

#	決定手続き[*15]	論理式[*16]	状態数[*17]	計算結果	実行時間[*18][s]	
					変更前	変更後
1	SAT	$2 \leq x+y \wedge x \leq 2 \wedge y \leq 2 \wedge x+y = 1+4z$	35 (640)	UNSAT	0.016	0.008
2	SAT	$2 \leq 2x+y \wedge 5x \leq 4y \wedge 5y \leq 4x+5$	70 (660)	UNSAT	0.008	0.004
3	SAT	$2 \leq x+y \wedge 3x \leq 6+y \wedge 3y \leq 6+x \wedge x+y = 1+4z$	156 (4000)	UNSAT	0.068	0.028
4	SAT	$y \leq 2x \wedge 12 \leq 3x+4y \wedge 5x+y \leq 15 \wedge y = 3z$	277 (10560)	SAT	0.116	0.056
5	SAT	$5 \mid x$	6 (2^8)	SAT	0.024	0.008
6	SAT	$10 \mid x$	8 (2^{13})	SAT	0.124	0.076
7	VALID	$2 \mid x \wedge 3 \mid x \Leftrightarrow 6 \mid x$	8 (2^{40})	VALID	N/A	0.036
8	VALID	$3a = b+c+d \wedge 3b = a+c+d \wedge 3c = a+b+d \wedge 3d = a+b+c \Leftrightarrow a = b \wedge a = c \wedge a = d$	262 (2^{36})	VALID	N/A	0.168

ることが分かる．SSReflect の行列積は素朴な 3 重ループによる定義なので，抽出されたプログラムの時間計算量は最善でも $\mathcal{O}(n^3)$ と考えられる．したがって，上記実験結果は，本研究のライブラリが余計なオーバヘッドを生じないことを示している．

7.2　プレスバーガー算術の決定手続き

より現実に即した問題として，**プレスバーガー算術**（*Presburger arithmetic*）の決定手続きがどの程度高速化できるかを調べた．プレスバーガー算術とは，自然数の加算に関する一階の理論であり，命題の充足可能性や妥当性が決定可能であることが知られている．また，プレスバーガー算術の決定手続きは Coq 上の自動証明手続きとしても使われている（`omega`，`lia` [2] など）．ここで用いる決定手続きは，プレスバーガー算術の命題を有限オートマトンに変換し，そのオートマトンの性質を調べることで，元の命題の充足可能性，妥当性が計算できる [4], [9] というものである．この決定手続きの形式化には，Doczkal らによる SSReflect ベースの正規言語ライブラリ [8] を用いた．この決定手続きや形式化の詳細についてはここでは説明しないが，文献 [21] で詳しく解説している．

この決定手続きでは，自由変数を n 個持つ命題を 2^n の語を受理するオートマトンに変換する．2^n は有限ドメイン関数 `bool ^ n` で表現され，本研究の手法はこの部分に対して作用する．また，有限オートマトンの状態遷移関数には有限ドメイン関数を使っていない．

本適用例については，ライブラリの差し替えにともなって高速化のための書き換えを行った．その書き換えの内容は以下のとおりであり，それ以外の変更はいっさい行っていない．

(1) 上限・下限を入れた整数の型 `range` に対する有限型のインスタンスを，`BijOrdMixin` を用いて再定義した．
(2) 定義中の有限型の濃度 `#|T|` の 2 カ所の出現を `$|T|` で置き換え，その定義を用いる証明を修正した．

特に後者の証明の修正は，2 カ所の `rewrite` の列に `-cardT'` を挿入するだけの非常に簡単な変更で済んでいる．

実験結果を**表 1** に示す．この実験結果を観察すると，まず 1〜6 番の場合については約 1.6〜3 倍の高速化が確認できる．また，7，8 番のように状態数が極端に多いものは変更前ではスタックオーバフローで計算できないところ，変更後では計算できるようになっている．よって，この実験の範囲では計算時間の短縮に加え，計算に必要な記憶領域を減らすような改善が実現できていると考えられる．

8.　モジュール性

本研究の特色として，変更前の SSReflect ライブラリを本研究で改良したライブラリに差し替えた際に，証明の変更が限定的な場合を除いて不要であること，すなわちライブラリのモジュール性があげられる．本章では，ここまでの内容を振り返りつつ，どのようにしてこのモジュール性を実現しているかをまとめる．また，このモジュール性が現実の証明に対してどの程度有効であるかを，具体例を通して検討する．

8.1　定義・補題の再現

ライブラリに変更を加えるうえで，元々あった定義・補題を提供できないということがあると，それらの定義・補題を使っていた証明は再度証明する必要があるか，もしくは使えなくなる．よって，モジュール性を成り立たせるうえで，元々提供されていた定義・補題をすべて提供するのは必須である．本研究で開発したライブラリでは，SSReflect に元々あった定義・補題はすべて再構成できている．特に

[*15] SAT は充足可能性の決定手続き，VALID は妥当性（恒真性）の決定手続きである．
[*16] 定数 n と式 e について，$n \mid e$ は $\exists x.\, nx = e$ の略（ただし x は e に対してフレッシュな変数）であり，「e は n で割り切れる」を意味する．
[*17] カッコ内は到達不能な状態を含めた状態数である．本実験で用いたオートマトンの構成アルゴリズムは，到達不能な状態の除去，最小化などは行わないようになっている．
[*18] N/A と書いてある項目は，スタックオーバフローによって計算できなかったことを示している．

FinMixin が再現できている（3.3 節）ことによって，元々あった有限型のインスタンスがすべて使えるようになっているのは重要な点である．

8.2　重要な定義の隠蔽

SSReflect の有限型・有限ドメイン関数ライブラリでは，多くの重要な定義が展開できないようにロックされ，それによってライブラリのモジュール性が守られている（4.1 節）．本研究での定義の変更の範囲は，ほとんどの部分がロックされた定義に限定される：

(1) Finite モジュール内の有限型の定義（3.2 節）
(2) 有限型の各インスタンス（3.4 節）
(3) 有限型に関連するロックされた計算の定義（4 章）
(4) ord_enum の定義（3.1 節）

このうち，(1)〜(3) については有限型に関連する定義のアンロックを行わなければ証明には影響せず，(4) は通常は直接使われない（代わりに enum 'I_n を用いる）定義である．よって，SSReflect を使って書かれた証明は定義のアンロックさえしていなければ変更なしで通るようになっている．

8.3　高速化のための書き換え

上述のように，SSReflect を用いて書いた証明はごく一部の例外を除いて変更なしに本研究のライブラリでも利用できる．しかし，本研究の手法を十分に生かした高速化をするためには，

- 有限型の各インスタンスを ordinal 型との間の全単射を用いて再定義し，
- 有限型 T の濃度 #|T| を $|T| で置き換える（4.2 節），

必要がある．

しかし，これらの変更が既存の証明に及ぼす影響は非常に少なく済む．前者については，有限型のインスタンスの中身を直接使う重要な定義がすべてロックされていることから明らかである．後者については，補題 cardT' を用いて $|T| を #|T| に書き換えることによって元の証明をそのまま利用できる．

8.4　現実の証明に対する有効性

本研究でのモジュール性を成り立たせるための工夫が，現実の証明に対してどの程度有効であったかを述べる．まず，7 章で示した適用例のうち，行列の積（7.1 節）については変更は不要であった．一方，プレスバーガー算術の決定手続きについては，高速化のための書き換えを行った．その書き換えの内容は，7.2 節で示したとおりの非常に簡単なものである．

また，我々は本研究で導入した新しい有限型・有限ドメイン関数の定義に合わせて Gonthier らの Feit-Thompson 定理の形式証明 [12] の検査を通すのに必要な書き換えの量を調べた．この Feit-Thompson 定理の形式証明は約 17 万行の Coq のコードからなる非常に巨大な形式証明[*19]であり，ライブラリのモジュール性を測る指標としても役立つと考えられる．我々は，有限型・有限ドメイン関数ライブラリを差し替えたうえでさらに 10 行以下の変更をすることで，Coq 8.5pl2 [*20]上で Feit-Thompson 定理の形式証明の検査を再度通すことに成功した．

我々は問題を単純にするため，本研究で用いている SSReflect 1.6 との差分が小さい Feit-Thompson 定理の形式証明を用いる必要があった．しかしながら，Feit-Thompson 定理の形式証明の正式なリリースは最も新しいものでも 2013 年という非常に古いものであったため，Mathematical Components の Git リポジトリ [20] の，SSReflect 1.6 に相当するコミットの直前[*21]にある Feit-Thompson 定理の証明を用いることにした．

有限型・有限ドメイン関数ライブラリ以外でどのような変更を行ったかを，以下にまとめる．

ssreflect/bigop.v　総和・総乗などを一般化した演算を扱うためのライブラリ．2 カ所で enum の定義をアンロックしており，それらの証明の修正が必要となった．

ssreflect/finset.v　有限冪集合型ライブラリ．finfun を用いて有限型の冪集合型を定義している．有限集合の冪集合は有限集合であり，これに相当する有限型のインスタンスが定義されているため，本研究の手法に合わせて（BijOrdMixin を用いて）再定義した．
なお，この変更は有限冪集合に関する計算の高速化のために行ったものであり，証明の検査を通すだけであれば不要である．

ssreflect/binomial.v　二項係数（binomial coefficients）ライブラリ．2 カ所で enum の定義をアンロックしており，それらの証明の修正が必要となった．

solvable/burnside_app.v　可解群のバーンサイドの定理に関するライブラリ．1 カ所で符号化関数の定義をアンロックしており，その証明の修正が必要となった．

odd_order/stripped_odd_order_theorem.v　Coq 8.5beta1 以降で Require コマンドがライブラリの完

[*19] SSReflect は Feit-Thompson 定理の形式証明を行った研究グループによって開発されており，Feit-Thompson 定理の形式証明は SSReflect 込みで配布されている．約 17 万行の Coq コードのうち，約 10 万行はそれと別に配布されている SSReflect ライブラリと共通である．逆に，SSReflect ライブラリの正式なリリースは Feit-Thompson 定理の形式証明から不要なファイル群を取り除くことで作られている．

[*20] 本研究で主に用いている Coq 8.5beta3 では，変更前の Feit-Thompson 定理の形式証明のうち field/finfield.v の検査に失敗したため，この証明の検査のみ Coq 8.5pl2 を用いることとした．この問題の内容は，証明チェッカ coqtop が計算機のメモリを使い果たすというもので，我々は Coq 自体のバグが原因であると考えている．また，SSReflect ライブラリの範囲に限れば同様の変更で Coq 8.5beta3 でも検査が可能であった．

[*21] ハッシュ値は 7c7309ad66db9fa2113edef6cba85dea4cc6c0cc である．

全な名前を要求するようになったことで検査に通らなくなっていたため，修正した．この問題はライブラリの差し替えとは無関係であり，元から修正が必要だった部分である．

これらの変更は，証明の検査を通すためだけであれば不必要な `finset.v`，今回のライブラリの差し替えに無関係な `stripped_odd_order_theorem.v` を除くと，合計で 6 行のコードの削除，8 行のコードの挿入で済んでいる．ここでの行数は，たとえば 1 行の編集であれば 1 行の削除と 1 行の挿入に分けて曖昧性のないように数えているが，実際には「8 行程度の編集」であるといえる．このように変更がほとんど不要であったことから，モジュール性に関する SSReflect ライブラリと本研究の手法は非常に優れていると考えられる．

9. おわりに

本研究では，有限型を `ordinal` 型との全単射を用いて再定義するという非常に簡単なアイディアに基づいて，既存ライブラリを書き換え，証明から抽出されたプログラムの効率を大きく改善できることを実例を通じて示した．本手法の特長は，既存の SSReflect ライブラリを使った証明をほとんど書き換えることなく，改善したライブラリの証明とすることができる点である．これにより，ユーザは SSReflect を使った証明から，スムーズに移行することができるという利点がある．SSReflect は，Coq の「代替標準ライブラリ」として非常に広い範囲で使われており，多くの Coq ユーザが本研究の恩恵を受けられるものと考えている．

本手法の効率改善の効果について，行列積およびプレスバーガー算術の決定手続きの証明で実証した．特に，後者のようなサイズの大きな証明を，本研究のライブラリを使って再検証し，プログラム抽出により実際に動作する OCaml コードを抽出できたということは，本研究が机上のアイディアだけでなく，大規模な証明で実際に利用可能なライブラリであることを示すものであると考える．

最後に，関連研究と今後の課題を述べる．

9.1 関連研究

Coq 上での定義やプログラム抽出での定義の読み替えを工夫して高速なプログラムを抽出する研究としては，冪集合とビット列 [3]，簡潔データ構造 [18]，二分決定図 [5] に関するものがあげられる．この中でも特に Blot らによる冪集合とビット列に関する高速なプログラムの抽出は，本研究の手法でも冪集合とビット列に関する計算を扱えること，本研究と同様に SSReflect の有限型ライブラリを使っていることを考えると非常に近いといえる．

Blot らの手法の優れた点は，冪集合をビット列を用いて計算機上で効率的に扱う方法に着目し，非常に効率の良いコードを抽出した点にある．この方法を用いて高速化された n-クイーン問題の解の探索は，OCaml や C で実装したものとほぼ同等の性能を出している．Blot らの手法と比較した我々の手法の利点は，より一般的な対象である有限ドメイン関数に着目しており，適用範囲が広いことである．

本研究の応用例として行列積の計算の高速化をあげているが，Coq 上での行列演算の高速化に関する既存研究としては CoqEAL [6], [7] があげられる．CoqEAL では SSReflect の行列をリストのリストに読み替えることで計算に使えるようにしたうえで[*22]，Winograd のアルゴリズムを用いて高速な行列積の計算を実現している．また，行列だけではなく多項式についても，カラツバ法を用いた積の計算の高速化に取り組んでいる．

CoqEAL では Coq 上での計算の高速化に焦点を当てており，代数に関する個々の効率的なアルゴリズムの実装と，素朴な定義と効率的な定義の同等性を示す手法の開発に取り組んでいる．一方，本研究では抽出されるプログラムの高速化に焦点を当てており，有限性に関する一般的な構造に着目した高速化に取り組んでいる．なお，計算機や実行方法が異なるため実験結果は単純には比較できないが，CoqEAL では 200×200 行列の積の計算に `vm_compute` で約 25 秒 [7] Fig. 2 かかっており，一方我々の方法では 6.208 秒となっている．よって，計算時間については我々の手法の方が優れているか，もしくは同程度であると考えられる．

9.2 今後の課題

9.2.1 抽出されたコードの解析・プロファイリング

SSReflect ライブラリが非常に多くの定義を含んでいるため，本研究の手法で抽出されたプログラムは非常に大きく，そのプログラムの構造を理解するのは困難である．そのため，本研究のライブラリを用いて抽出されたプログラムにはまだ高速化の余地があると考えている．そこで，抽出されたコードを解析・プロファイリングし，さらなる最適化の余地があるのかをより深く検討する必要がある．

9.2.2 他言語のプログラム抽出への拡張

本研究では抽出先の言語として OCaml のみを用いたが，Coq では OCaml 以外にも Haskell や Scheme のプログラム抽出に対応しており，他にも非公式ではあるが Ruby や Scala のプログラム抽出にも対応している．本研究の手法は OCaml 特有の機能を用いるものではないため，他の言語に対しても同様に適用可能であると予想している．この予想の真偽を明らかにし，本手法がどの程度広い範囲まで適用可能かを示すため，他の抽出先言語についても 7 章と同様の実験を行う必要がある．

9.2.3 部分型を用いて定義される有限型への対応

本研究で高速化の鍵となるのは，有限型と `ordinal` の間

[*22] 多数の定義が隠蔽されているため，SSReflect の行列そのままでは Coq 上で計算を行うのは困難である．

の全単射であった．3.4，5.2 節で説明したように，直和，直積，有限ドメイン関数などの多くの標準的な有限型については効率の良い全単射が構成できている．しかしながら，有限型の部分型（有限対称群など）については，個々の場合では効率の良い全単射を構成できることもあるが，一般的には数え上げによってしか構成できない．

そこで，有限型の部分型をドメインとする有限ドメイン関数について，派生元の型の全単射を用いてインデックスの計算を行う選択肢を提供するなど，本研究の手法を有限型の部分型に拡張するための工夫が必要であると考えている．

謝辞　プログラミング研究会での質疑と匿名査読者によるコメントは，本研究の内容と本論文で伝えるべき内容を再検討するうえで，非常に役立つ内容であった．質問をいただいた方々と査読者に感謝する．本研究に関するコメントをいただいた海野広志先生（筑波大学）に感謝する．本研究は，科学研究費（15K12007）の補助を受けている．

参考文献

[1] Bertot, Y., Gonthier, G., Ould Biha, S. and Pasca, I.: Canonical Big Operators, *Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science, Vol.5170, pp.86–101, Springer (2008).
[2] Besson, F.: Fast Reflexive Arithmetic Tactics the Linear Case and Beyond, *Types for Proofs and Programs*, Lecture Notes in Computer Science, Vol.4502, pp.48–62, Springer (2007).
[3] Blot, A., Dagand, P.-É. and Lawall, J.: From Sets to Bits in Coq, *Functional and Logic Programming*, Lecture Notes in Computer Science, Vol.9613, pp.12–28, Springer (2016).
[4] Boudet, A. and Comon, H.: Diophantine equations, Presburger arithmetic and finite automata, *Trees in Algebra and Programming – CAAP '96*, Lecture Notes in Computer Science, Vol.1059, pp.30–43, Springer (1996).
[5] Braibant, T., Jourdan, J.-H. and Monniaux, D.: Implementing Hash-Consed Structures in Coq, *Interactive Theorem Proving*, Lecture Notes in Computer Science, Vol.7998, pp.477–483, Springer (2013).
[6] Dénès, M.: CoqEAL—The Coq Effective Algebra Library, available from ⟨http://www.maximedenes.fr/content/coqeal-coq-effective-algebra-library⟩.
[7] Dénès, M., Mörtberg, A. and Siles, V.: A Refinement-Based Approach to Computational Algebra in Coq, *Interactive Theorem Proving*, Lecture Notes in Computer Science, Vol.7406, pp.83–98, Springer (2012).
[8] Doczkal, C., Kaiser, J.-O. and Smolka, G.: A Constructive Theory of Regular Languages in Coq, *Certified Programs and Proofs*, Lecture Notes in Computer Science, Vol.8307, pp.82–97, Springer (2013).
[9] Esparza, J.: Automata Theory: An Algorithmic Approach (2016), available from ⟨https://www7.in.tum.de/~esparza/automatanotes.html⟩.
[10] Garillot, F., Gonthier, G., Mahboubi, A. and Rideau, L.: Packaging Mathematical Structures, *Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science, Vol.5674, pp.327–342, Springer (2009).
[11] Gonthier, G.: Point-Free, Set-Free Concrete Linear Algebra, *Interactive Theorem Proving*, Lecture Notes in Computer Science, Vol.6898, pp.103–118, Springer (2011).
[12] Gonthier, G., Asperti, A., Avigad, J., Bertot, Y., Cohen, C., Garillot, F., Le Roux, S., Mahboubi, A., O'Connor, R., Ould Biha, S., Pasca, I., Rideau, L., Solovyev, A., Tassi, E. and Théry, L.: A Machine-Checked Proof of the Odd Order Theorem, *Interactive Theorem Proving*, Lecture Notes in Computer Science, Vol.7998, pp.163–179, Springer (2013).
[13] Gonthier, G., Mahboubi, A. and Tassi, E.: A Small Scale Reflection Extension for the Coq system, Research report, INRIA (2015), available from ⟨https://hal.inria.fr/inria-00258384v16⟩.
[14] Letouzey, P.: A New Extraction for Coq, *Types for Proofs and Programs*, Lecture Notes in Computer Science, Vol.2646, pp.200–219, Springer (2003).
[15] Mahboubi, A. and Tassi, E.: Canonical Structures for the Working Coq User, *Interactive Theorem Proving*, Lecture Notes in Computer Science, Vol.7998, pp.19–34, Springer (2013).
[16] Paulin-Mohring, C.: Extracting F_ω's programs from proofs in the Calculus of Constructions, *16th Annual ACM Symposium on Principles of Programming Languages*, Austin, ACM (1989).
[17] Sakaguchi, K.: The Coq development accompanying this paper, available from ⟨http://logic.cs.tsukuba.ac.jp/~sakaguchi/src/efficient-finfun-ipsj-pro-2016-1.tar.gz⟩.
[18] Tanaka, A., Affeldt, R. and Garrigue, J.: Formal Verification of the rank Function for Succinct Data Structures, *Proc. 18th JSSST Workshop on Programming and Programming Languages* (2016).
[19] The Coq Development Team: *The Coq Proof Assistant Reference Manual* (2015), available from ⟨https://coq.inria.fr/distrib/8.5beta3/refman/⟩.
[20] The Mathematical Components project: The Mathematical Components repository, available from ⟨https://github.com/math-comp/math-comp⟩.
[21] 坂口和彦：Coq による定理証明—2015.12，Tsukuba Coq Users' Group (2015)，入手先 ⟨http://tcug.jp/books/2015-12/⟩.

坂口 和彦　（学生会員）

2014 年筑波大学情報学群情報科学類卒業．同年筑波大学大学院システム情報工学研究科コンピュータサイエンス専攻進学．2013 年より，Tsukuba Coq Users' Group で定理証明器に関連する書籍の執筆・発行に携わる．定理証明器のソフトウェア検証・プログラム言語・数学への応用に興味を持つ．

亀山 幸義　（正会員）

筑波大学大学院システム情報工学研究科コンピュータサイエンス専攻教授．プログラム論理と検証・関数型プログラム言語・段階的計算に興味を持つ．ACM，日本ソフトウェア科学会各会員．