日本ソフトウェア科学会第 26 回大会 (2009 年度) 講演論文集

プログラム運算のための Coq ライブラリ

橋本 英樹　胡 振江　Julien Tesson　Frédéric Loulergue
武市 正人

プログラム運算は，プログラムを，その意味を変えない運算規則 (定理) を用いて，より効率のよいプログラムに書き換えていくプログラミング手法である．プログラム運算のよい特徴として，運算前のプログラムが自明に正しいプログラムであれば，運算後のプログラムも意味を保存しているため検証が不要であり，正しく効率のよいプログラムの開発に利用しうることがあげられる．本研究で，我々は定理証明支援系 Coq のライブラリとしてプログラム運算を支援するシステムを構築した．これを用いて，ユーザーは証明された定理のみを用いて安全にプログラム運算を行うことができる．また，我々は応用として，Bird によるプログラム運算のレクチャーノートを表現した．リスト上の関数のための運算規則を証明し，プログラム運算を我々のライブラリを用いて Coq で表現した．本発表では，上記のプログラム運算支援システム，および応用例の実装を紹介する．

1 はじめに

1.1 背景

プログラム運算とは，プログラムを，その意味を変えない運算規則 (定理) を用いて，より効率のよいプログラムに書き換えていくプログラミング手法である [8]．運算前のプログラムが，もし疑いの余地なく正しいプログラムであれば，運算後のプログラムも，元々のプログラムと同じ意味を持っているので，正しいプログラムとなる．このことを活かし，プログラムの簡潔さと効率を両立させられる，というのがプログラム運算を用いた開発の利点である．また，プログラマが意図していなかった，よりよいアルゴリズムが導出されることも考えられる．

プログラム運算の過程が長い場合，人手によるプログラム運算では間違いが入りこむ可能性が高く，また，正確に運算規則が適用されているかを確認するのが難しい．そこで，運算規則が正しく適用できるか確認でき，運算規則自体が正しいことを証明でき，さらに自動的に運算規則の適用を行えるようなシステムが望まれ，発展されてきた [10] [6] [11] [7] [8] [5].

1.2 本研究の貢献

本研究では，定理証明支援系 Coq [1] において，証明されたプログラム運算規則のみを用いて安全にかつ対話的にプログラム運算を行えるシステムを簡単に構築できることを示した．具体的には，推論を伴う等式変形を自然に記述するための tactic および記法を Coq ライブラリとして実装した．また，そのライブラリの効果を確認するために，プログラム運算の例として Bird によるプログラム運算のレクチャーノートである “An introduction to the theory of lists” [3] (以下 Theory of Lists と略記する) を取り上げ，このライブラリを用いて表現した．その中で，プログラム運算をより読みやすく，自然に記述するための知見を得た．この知見についても報告する．

A Coq Library for Program Calculation.

Hideki Hashimoto, Masato Takeichi, 東京大学大学院情報理工学系研究科, Graduate School of Information Science and Technology, The University of Tokyo.
Zhenjiang Hu, 国立情報学研究所, National Institute of Informatics.
Julien Tesson, Frédéric Loulergue, The University of Orleans.

1.3 関連研究

プログラム運算，またはより広くプログラム変換を支援するシステムは Kids [10]，MAG [6]，Stratego [11]，Ultra [7]，Yicho [8]，RAPT [5] など多数存在する．

さらに最近，Mu ら [9] は，本研究と同様に，依存型をもつプログラミング言語 Agda を用いて証明可能なプログラム運算を記述した AoPA ライブラリを提案した．しかし，Agda は自動証明を行うための機能を持たず，プログラムを書き換えるという観点から手間がかかるという欠点があった．

本研究は，定理証明支援系 Coq において証明の構成法をプログラミングし，tactic を追加するための言語 Ltac を用いることにより，証明された運算規則のみを用いて安全にプログラム運算を行えるシステムを構築できることを示した．この方法のよい点として，tactic が自動証明などに容易に拡張できること，さらに既存の tactic と併用できること，既存の Coq ライブラリが活用できること，対話モードによって試行錯誤的な使い方ができることなどが挙げられる．

1.4 本稿の構成

本稿の構成は以下の通りである．まず，2 節で Coq の諸概念と記法について述べる．次に，3 節で我々が実装した rewriting ライブラリについて紹介する．続いて 4 節で Theory of Lists に現れる諸定理およびプログラム運算を実装した際の工夫について述べる．最後に，5 節で，まとめと今後の課題について述べる．

2 Coq の概要

Coq [1] [2] は INRIA (フランス国立情報学自動制御研究所) によって開発されている定理証明支援系である．以下，Coq の諸概念，および表記法について簡単に説明する．詳細は教科書 [2] 等を参照されたい．

2.1 セクション，型定義，記法定義

以下は Coq でのリストの定義である．

```
Section list.
Variable A : Type.
Inductive list : Type :=
  | nil : list
  | cons : A -> list -> list.
End list.
```

1 行目の `Section` コマンドは，新しくセクションをつくる．セクションの中で定義された宣言や定義は， セクション中の，その宣言や定義よりも後でのみ有効である．2 行目の `Variable` コマンドは，それを囲む最も内側のコンテクストでのみ有効な変数を宣言する．ここでは，`Type` 型の変数 `A` を宣言している．`A` が `Type` 型をもつというのは，`A` が型だという意味である．3 行目の `Inductive` で，新たな型を定義する．ここでは `list` を `Type` 型をもつ型として定義している．4 行目と 5 行目の “|” 以降が `list` のコンストラクタで，それぞれ `list` 型をもつ `nil`，`A -> list -> list` 型をもつ `cons` であると定義している．6 行目の `End` コマンドでセクションを終了する．セクションを終了すると，セクションの内側で宣言されたローカルなスコープをもつ変数を参照することができなくなる．それに伴い，セクションの中で定義されたグローバルな定義が，ローカルな変数に依存している場合，その定義はローカルな変数だったものを引数にとる関数となる．たとえば，`list` の定義はローカルな変数 `A : Type` に依存しているので，`End` コマンドで `list` セクションを閉じた後では，`list` の型は `Type -> Type` に，コンストラクタ `nil` の型は `forall (A : Type), list A` に，それぞれ変化する．

Coq では，`Notation` コマンドなどのコマンドによって，新たな記法を導入することができる．以下は，`list` のコンストラクタである `cons` を，中置演算子 “::” で表現できるようにするための `Infix` コマンドの例である．

```
Infix "::" :=
  cons (at level 60, right associativity)
  : list_scope.
```

2.2 関数定義

関数を定義するには，`Fixpoint` コマンドを用いる．以下は，2 つのリストを引数にとり，それらを連結し

```
Section fold.
Variables A B : Type.
Fixpoint foldr (op : A -> B -> B) (e : B) (x : list A) : B :=
  match x with
    | nil => e
    | a :: x' => op a (foldr op e x')
  end.

Fixpoint foldl (op : B -> A -> B) (e : B) (x : list A) : B :=
  match x with
    | nil => e
    | a :: x' => foldl op (op e a) x'
  end.
End fold.
```

図 1　foldr および foldl の定義

た新しいリストを返す関数 append を定義する例である．

```
Section append.
Variable A : Type.
Fixpoint append (x y : list A)
                {struct x} : list A :=
  match x with
    | nil => y
    | a :: x' => a :: (append x' y)
  end.
End append.
```

3 行目の append が関数名であり，それに続く x, y が引数である．“: list A” の部分が，結果の型が list A 型であることを表す．関数の定義の中で，match 構文を使って，引数に関するパターンマッチを行っている．すなわち，この関数 append は，第 1 引数 x が nil の場合は y を返し，a :: x' の場合は a を再帰呼び出し append x' y の結果の先頭に追加する関数である．以上のように，Fixpoint コマンドを用いて，再帰的な関数を定義することができる．

同様にして高階関数も定義できる．図 1 に以下のように定義される高階関数 *foldr* および *foldl* の Coq での定義を示す．

$$
\begin{aligned}
&\mathit{foldr}\ (\oplus)\ e\ [a_1, a_2, \ldots, a_n] \\
&\quad = \quad a_1 \oplus (a_2 \oplus \cdots \oplus (a_n \oplus e)) \\
&\mathit{foldl}\ (\oplus)\ e\ [a_1, a_2, \ldots, a_n] \\
&\quad = \quad ((e \oplus a_1) \oplus a_2) \oplus \cdots \oplus a_n
\end{aligned}
$$

2.3　命題の表現と tactic による証明

ここでは，命題を表現する方法について述べる．Coq においては，Curry-Howard 対応に基づいて，命題は型であり，証明は関数である．したがって，Coq で新たな推論規則を導入するためには，推論規則を反映した新たな型を定義すればよい．公理を導入するには，Axiom コマンドによってグローバルな宣言をする．

Coq が採用している型システムは，依存型を含んでいる．例えば，2 つの項が同一であるという命題 eq は，Coq では依存型を用いて次のように定義されている．

```
Inductive
  eq (A : Type) (x : A) : A -> Prop :=
    | refl_equal : eq x x
```

Prop は，Type と同じく，型につく型である．標準ライブラリでは，eq のための記法が用意されており，x = y は eq x y と同じ意味である．この表記を用いると，1 + 1 = 2 は型であり，たとえば refl_equal 2 がこの型をもつ．すなわち，refl_equal 2 は 1 + 1 = 2 型をもつ．このことを，以下の記述で確認することができる．

```
Definition oot : 1 + 1 = 2 := refl_equal 2.
```

Definition コマンドは，Fixpoint コマンドと同様，グローバルな定義を与える．ここでは，oot という識別子に，refl_equal 2 という定数を割り当て，それが 1 + 1 = 2 型をもつことを確認している．なお，Definition コマンドで型のみを示し定義を示さない場合，tactic による証明を行う状態 (対話モード) に

なる．対話モードでは，証明すべき命題すなわち型がゴールとして示され，また現在のコンテクストに含まれる宣言および定義が前提として列挙される．ユーザーは tactic によって，現在の前提からどのように証明を構成するかを指示する．例えば, 上記の `oot` の場合には，以下のように tactic で証明を完成させることができる．

```
Definition oot' : 1 + 1 = 2.
  reflexivity.
Qed.
```

ここでは `refl_equal` を用いて証明を完成させる tactic である `reflexivity` を用いて，証明を構成した．証明が完了した状態で `Qed` コマンドを入力すると，対話モードで構成された証明が，定義として環境に登録される．

2.4 関数の外延的等価性

関数の外延的等価性とは，2 つの関数が任意の入力に対して同じ値を返すときその 2 つの関数が等しい関数であるとみなすという概念である．本研究では外延的に等価な関数をプログラム運算によって導出することを意図しているため，我々は次の一つの公理を導入した．†1

```
Axiom fun_ext :
  forall (A B : Type) (f g : A -> B),
    (forall x : A, f x = g x) -> f = g.
```

`Axiom` コマンドは，公理をグローバルに定義するものである．「任意の関数 `f` と `g` に対して，それぞれの出力が定義域 `A` 全域で等しければ，`f` と `g` は等しい」という命題を型で表現し，`fun_ext` という識別子はその型を持つとしている．

3 等式推論記述のための tactic ライブラリ

3.1 Coq でのプログラム運算

プログラム運算支援システムは以下のような性質を持つことが望ましい．第一に，創造的なプログラムの導出がスムーズに行われるよう，対話的にプログラム運算が行えることが望ましい．ユーザーが意図した大きな運算ステップは自動的に行われることが望ましいものの，完全な自動運算は一般には難しい．第二に，プログラム運算で得られたプログラムは，当初の仕様を満たしているという意味で正しいプログラムであることが保証される必要がある．第三に，新たなプログラム運算規則を追加できる必要がある．なぜなら全ての運算規則を網羅することは不可能だからである．最後に，プログラム運算の過程が保守しやすい形で保存され，文書化されることが望ましい．

Coq はプログラム運算を支援するための多くの機能を備えている．ユーザーは対話的に証明を行うことができ，依存型によって仕様，および運算規則を表現することができる．また，証明の構成は正しい運算と対応し，等式を用いてゴールの一部を書き換える効果をもつ “rewrite” tactic は式の操作に役立つ．しかし，実用上，Coq でプログラム運算を行にはいくつかの問題点がある．ひとつは，Coq は，ある実装が仕様を満たしているという方向の証明を行う tactic は豊富だが，仕様からよりよい実装を導出するという方向の tactic は乏しい．もうひとつは，証明スクリプトは読みづらく，保守しづらい．たとえば，標準ライブラリの以下の単純な定理 `appAssoc` の証明は Coq の知識無しには読みやすいものではない．

```
Lemma appAssoc:
forall (A: Type) (l m n: list A),
  (l ++ m) ++ n = l ++ (m ++ n)
Proof.
  intros. induction l.
  simpl in |- *; auto.
  change (a : (l ++ m) ++ n = a : l ++ m ++ n)
    in |- *.
  rewrite <- IHI; auto.
Qed.
```

3.2 等式推論記述のための tactic ライブラリの実装

前述の課題を解決するために，我々は，Coq でのプログラム運算を支援するための tactic ライブラリである rewriting ライブラリを開発した．このライブラリの主な構成要素を以下順を追って説明する．

†1 関数の外延的等価性は標準ライブラリの `Coq.Logic.FunctionalExtensionality` にて提供されているが, 不必要な定義などが含まれるため本研究では独自の公理を導入した．

3.2.1 運算状態の保持

rewriting ライブラリでは，以下のような状態を表す型 state を用いて，現在の運算のフォーカスを表現することができる．

```
inductive state : Prop := RHS : state
                        | LHS : state
                        | GOAL : state
```

これらの値をコンテクストに追加することで運算の状態を表す．例えば、LHS t と，tactic の前に LHS を付加して tactic t が呼びだされたら，コンテクストに LHS を追加し，現在左辺を等式変形しているということを表現する．さらに，その後の tactic で状態に応じた条件分岐をすることもできる．

3.2.2 等式変形の理由づけ

我々は，簡潔に等式変形が記述できるように次の tactic を用意した．

```
= { t }
  e
```

この tactic は，コンテクストに LHS が登録されている場合，以下の Ltac として解釈される．

```
match goal with
  [|- ?lhs = ?rhs] =>
    let h := fresh "rewriting" in
      assert (h : lhs = e) by
        (t; reflexivity); rewrite h
end
```

これは，tactic t を実行することによって現在のゴールの左辺と e が等しくなることを新たに証明し，それを用いてゴールを書き換えている．この tactic を連ねて記述することで，等式変形を行うことができる．例として，前述の定理 appAssoc をこの tactic を用いて証明したものを図??に示す．

3.2.3 tactic 名の別表記

先述の={ t } e tactic の t としては任意の tactic を使うことができるが，より記述を読みやすくするため，以下のような記法を導入した．by def は定義を展開する tactic である unfold の別名である．pat は左辺または右辺のどの部分式を書き換えるかを指定する tactic であり，refine tactic を用いて実現されている．さらに，証明の終了には， reflexivity の別表記として，“[]”が使えるようにした．

```
Lemma appAssoc:
forall (A: Type) (l m n: list A),
  (l ++ m) ++ n = l ++ (m ++ n).
Proof.
Begin.
induction l.

check ((nil ++ m) ++ n = nil ++ m ++ n) is GOAL.
      LHS
    =  { by def app }
      (m ++ n).
    =  { by def app }
      RHS.
    [].

check (((a :: l) ++ m) ++ n =
  (a :: l) ++ m ++ n) is GOAL.
      LHS
    =  { by def app }
      ((a :: (l ++ m)) ++ n).
    =  { by def app }
      (a :: ((l ++ m) ++ n)).
    =  { rewrite IHl }
      (a :: (l ++ (m ++ n))).
    =  { by def app }
      RHS.
    [].
Qed.
```

図 2 rewriting ライブラリによる appAssoc の証明

3.2.4 exist 型の除去

次の定理 existEq のように，運算をした結果が分かっていない場合に，左辺のみを等式変形できる tactic を導入した．

```
Theorem existEq : exists n : nat, 1 + 1 = n.
Begin.
  LHS
={ by def plus }
  2.
[].
Qed.
```

Begin tactic はゴールの右辺に対応する変数を用意する．それ以降は先と同様に等式変形を記述することができ，望ましいものが得られたら，“[]”によって証明を完了させることができる．

4 Theory of Lists の実装

Theory of Lists [3] は Bird によるプログラム運算のレクチャーノートであり，プログラム運算の基本的なアイデアを平易に解説したものである．我々は，

前節で紹介した rewriting ライブラリを評価するために，このレクチャーノートに現れる定義を Coq で実装し，定理を証明し，そしてプログラム運算が証明としてうまく再現できることを確かめた．

Theory of Lists を実装する中で，その定義や証明を自然に表現するためにいくつかの工夫が必要になった．まず，部分関数を Coq の全域関数として表現する必要があった．また，リストを cons リストで実装しながら snoc リストや join リストとみなせるような仕組みが必要になった．また，*reduce* のように，引数となる関数に制約をもつ高階関数を表現する必要があった．

本節では，Theory of Lists について簡単に紹介し，Theory of Lists を実装する中で得られた，プログラム運算を記述する際の知見について説明する．

4.1 Theory of Lists の概要

Theory of Lists では，リストを走査する関数に関する定理と，それを用いた効率の良いプログラムの導出が示されている．このレクチャーノートの特徴的な点は，リスト走査の典型的なパターンが高階関数として抽出され，それらに対して BMF (Bird-Meetens Formalism) [3] と呼ばれる略記法が提供されていることである．これら高階関数に対して成り立つ運算定理を用意することにより，簡潔なプログラム運算が達成されている．

Theory of Lists では，リスト上の高階関数の代表的なものとして，例えば以下のものが定義され利用されている．“$*$” は *map* を表し，関数を第 1 引数にとり第 2 引数のリストの各要素に適用した新たなリストを返す．

$$f* \, [a_1, a_2, \ldots, a_n] = [f \; a_1, f \; a_2, \ldots, f \; a_n] \quad (1)$$

“$/$” は *reduce* を表し，結合的な 2 項演算を第 1 引数にとり第 2 引数のリストの各要素間に演算を挟みこんだ結果を返す．

$$\oplus/ \, [a_1, a_2, \ldots, a_n] = a_1 \oplus a_2 \oplus \cdots \oplus a_n \quad (2)$$

“$\triangleleft$” は *filter* を表し，*true* または *false* を返す述語を第 1 引数にとり，第 2 引数のリストの要素のうち，述語が *true* を返す要素のみからなるリストを返す．Theory of Lists では，このような高階関数を用いて，リスト上のさまざまな関数が定義できること，高階関数同士の等価性を表す定理が知られていること，それらの定理を用いてプログラム運算を行うことで効率化が図れることなどが紹介されている．

我々は，実際に Theory of Lists の関数を定義し、rewriting ライブラリに基づいてプログラム運算を行った．図 2 に，rewriting ライブラリを用いてプログラム運算を記述した例を示す．なお，比較のために，Bird の記法に基づいた証明も併記した．図 2 から見てとれるように，証明を等式変形のように記述することができている．

4.2 部分関数の扱い

Coq では，関数は証明の整合性を保つため，全ての関数は全域で，計算が停止しなければならない．したがって，部分関数を定義することはできない．しかし，プログラミングをする上で，部分関数を扱いたくなる場合は多い．たとえば，リストを取り，その先頭を返すような関数は，リストが空の場合に定義されないので，部分関数である．このような関数を，全域関数で表現する必要がある．

プログラム運算でプログラムを書き換えて他のプログラムに変換する，という目的から，部分関数を全域関数で表現する方法は，統一した方が望ましい．我々は，その方法として，部分関数に一つ引数を追加する方法を採用した．具体的には，その引数の型を値域の型とし，もし部分関数が未定義の場合は，その引数に渡された値を標準の値として出力とする．この方針に従うと，先述のリストの先頭を返す関数は，以下のように表現できる．

```
Section head.
Variable A : Type.
Fixpoint head (d : A) (x : list A) : A :=
  match x with
    | nil => d
    | a :: x' => a
  end.
End head.
```

ここでは，`d` を標準の値として，入力が本来関数が定義されない `nil` の場合は `d` を出力している．なお，この定義は，標準ライブラリの `hd` と同じものである．

```
Theorem filter_promotion :
  p <| :o: @concat A
      = @concat A :o: p <| *.
Proof.
    LHS
={ rewrite (filter_mapreduce) }
   (++ / :o: f * :o: @concat A).
={ rewrite map_promotion }
   (++ /:o: @concat (list A) :o: f* *).
={ rewrite comp_assoc }
   ((++ /:o: @concat (list A)) :o: f* *).
={ rewrite reduce_promotion }
   (++ / :o: (++ /)* :o: f* *).
={ rewrite concat_reduce }
   (@concat A :o: (++ /)* :o: f* *).
={ rewrite map_distr_comp }
   (@concat A :o: (++ / :o: f *) *).
={ rewrite filter_mapreduce }
   (@concat A :o: (p <|) *).
[].
Qed.
```

Theorem (filter promotion).
$p\triangleleft \circ concat = concat \circ (p\triangleleft)*$

Proof.

$$
\begin{aligned}
& \text{LHS} \\
= \quad & \{\ p\triangleleft = +\!\!+/ \circ f* \text{ where } f\ a = (p\ a)\ ?\ [a] : [] \ \} \\
& +\!\!+/ \circ f* \circ concat \\
= \quad & \{\ \text{map-promotion law} \ \} \\
& +\!\!+/ \circ concat \circ f** \\
= \quad & \{\ \text{associativity of} \circ \ \} \\
& (+\!\!+/ \circ concat) \circ f** \\
= \quad & \{\ \text{reduce-promotion law} \ \} \\
& +\!\!+/ \circ (+\!\!+/)* \circ f** \\
= \quad & \{\ concat = +\!\!+/ \ \} \\
& concat \circ (+\!\!+/)* \circ f** \\
= \quad & \{\ \text{map distributes over} \circ \ \} \\
& concat \circ (+\!\!+/ \circ f*)* \\
= \quad & \{\ p\triangleleft = +\!\!+/ \circ f* \ \} \\
& concat \circ (p\triangleleft)*
\end{aligned}
$$

□

図 3　プログラム運算の例．(左) **rewriting** ライブラリによるもの，(右) **Bird** の記法によるもの．

4.3　データ型の異なる見方の利用

我々は Theory of Lists においてリストの様々な見方が現れることに注目した．

我々はリストの定義として，2.1 節で示した，`nil` または `cons` で構成される項であるという定義を用いた．この定義は，リストは，空リストの先頭に要素を 1 つずつ追加していったものであるという見方を反映している．(以下，この定義を cons リストと呼ぶ．) しかし，リストの見方には他にもいくつかの方法がある．例えば snoc リストは，リストの空リストの末尾に要素を 1 つずつ追加していったものであるという見方である．また join リストは，リストが空リストであるか，要素が 1 つだけのリストであるか，2 つのリストを連結したものである，という見方である．

我々は，リストの実装を cons リストとしつつ，これらの見方を活用する方法について考察した．その理由は，関数同士の合成しやすさという観点からは，リストの実装が 1 種類である方が都合がよく，一方で，関数の定義や帰納法を用いた証明の記述において，リストを cons リストではなく，snoc リストや join リストであると考えた方が自然であるような場合が多々あるからである．実際，Theory of Lists では，cons リスト上の帰納法，snoc リスト上の帰納法，join リスト上の帰納法を用いた証明が全て現れている．

これら snoc リストおよび join リストを Coq で型として定義すると，それぞれ以下のようになる．

```
Section snoc.
Variable A : Type.
Inductive slist : Type :=
  | snil : slist
  | snoc : slist -> A -> slist.
End snoc.
Section join.
Variable A : Type.
Inductive jlist : Type :=
  | jnil : jlist
  | jsingleton : A -> jlist
  | jappend : jlist -> jlist -> jlist.
End join.
```

我々はリストを snoc リストや join リストとして定義した際に定義される帰納法を，cons リストで表現し，定理として証明した．また，snoc リストや join リストのための定義を利用して証明をするための定理を証明した．

まず帰納法の原理について述べる．2.1 節で `Inductive` コマンドを使って `list` を定義したが，その際に同時に `list` 型に関する帰納法の原理が Coq によって自動的に定義される．その項は `list_ind` と名づけられ，以下の型をもつ．

```
Section foldl_snoc.
Variables A B : Type.
Fixpoint foldl_snoc (op : B -> A -> B) (e : B) (x : slist A) : B :=
  match x with
    | snil => e
    | snoc x' a => op (foldl_snoc op e x') a
  end.
```

図 4　snoc を利用した foldl の定義

```
foldl_rev_char : forall (A B : Type) (op : B -> A -> B) (e : B) (f : list A -> B),
  f nil = e ->
  (forall (a : A) (x : list A), f (x ++ [a]) = op (f x) a) ->
  f = foldl op e
```

図 5　foldl_rev_char の定義

```
list_ind :
  forall (A : Type) (P : list A -> Prop),
  P nil ->
  (forall (a : A) (l : list A),
   P l -> P (a :: l)) ->
  forall l : list A, P l
```

同様に先ほどの slist 型および jlist 型を定義した場合にも帰納法の原理が定義されるはずである．我々はその帰納法の原理を cons リストで表現し，定理として利用した．ここにそれぞれの型を示す．

```
rev_ind :
  forall (A : Type) (P : list A -> Prop),
  P nil ->
  (forall (x : A) (l : list A),
   P l -> P (l ++ [x])) ->
  forall l : list A, P l

join_induction :
  forall (A : Type) (p : list A -> Prop),
  p nil ->
  (forall a : A, p ([a])) ->
  (forall x y : list A,
   p x /\ p y -> p (x ++ y)) ->
  forall x : list A, p x
```

rev_ind は標準ライブラリの Coq.List.Lists ライブラリに含まれているものを用いた．join_induction は新たに証明した．

次に，snoc リストに関して定義された関数の利用について述べる，2.2 節で，*foldl* という関数について述べた．この関数は，snoc リストに対して自然に定義される関数である．すなわち，上の slist の定義を使えば図 3 のように定義できる，cons リスト上の関数としてこのような定義をすることはできないので，我々は図 4 に示される型を持つ定理 foldl_rev_char を証明した．なお，x ++ [a] は，append x (cons a nil) を Notation コマンドによって導入した表記で表現したものである．この定義に現れる foldl は，2.2 節に現れるものと同じもので，cons リストについて定義された関数である．この定理は，ある関数 f が foldl op e と等しいことを示すために， f nil = e であること，および任意の a, x について f (x ++ [a]) = op (f x) a が成り立つことの 2 つを示せばよい，ということを表現している．この定理は，上の foldl_snoc 関数の定義を反映している．

4.4　演算の性質を強める手法

4.1 節で紹介した高階関数 *reduce* は，以下の性質をもつ．

$$\forall a,\ \oplus/[a] \quad = \quad a \tag{3}$$

$$\forall x\ y,\ x \neq nil \wedge y \neq nil \Rightarrow$$
$$\oplus/(x +\!\!+ y) \quad = \quad (\oplus/x) \oplus (\oplus/y) \tag{4}$$

直観的には，性質 (4) は，リストを任意の場所で分割して，独立に計算できるという性質を表現している．この性質を用いたプログラム運算ができれば，効率化に貢献すると期待される．ところが，*reduce* を単純に 2 引数関数と入力リストを引数として定義しようとすると，演算子が結合的であるという条件を考慮していないため，結合的でない演算子の *reduce* を定義してしまう危険性がある．その結果として，上記の性質が証明できることは期待できない．*reduce* の

性質を利用するために，我々は以下のように *reduce* を実装した．

```
Section reduce_mono.
Variables (A : Type)
          (op : A -> A -> A) (e : A).
Definition reduce_mono (m : monoid op e) :
  list A -> A :=
    foldl op e.
End reduce_mono.
```

この `reduce_mono` の定義は，Coq では証明が関数であることを利用している．`reduce_mono` の第 1 引数 `m` の型は，`monoid op e` である．この `monoid op e` 型は，`A` 上の二項演算子 `op` が結合的で，`e` がその単位元になっているという命題を依存型で表現したものである．そして，この関数は，`foldl op e` を返す．もし，`op` が結合的で，`e` がその単位元になっているとすると，`foldl op e` が *reduce* を表現していることは以下のように確かめられる．長さ 1 以上のリスト $[a_1, a_2, \ldots, a_n]$ について，

$$
\begin{aligned}
&\mathit{foldl}\ (\oplus)\ e\ [a_1, a_2, \ldots, a_n] \\
&\quad = \quad ((e \oplus a_1) \oplus a_2) \oplus \cdots \oplus a_n \\
&\quad = \quad e \oplus a_1 \oplus a_2 \oplus \cdots \oplus a_n \qquad (\oplus\text{の結合性より}) \\
&\quad = \quad a_1 \oplus a_2 \oplus \cdots \oplus a_n \qquad (e \text{ は } \oplus \text{ の単位元})
\end{aligned}
$$

となるからである．同様に，この定義によって，*reduce* の性質 (3) および (4) が満たされる．特に，この定義では x または y が *nil* の場合でも (4) が成立する．

しかし，一般に，結合的な演算子に単位元が存在するとは限らない．そこで先ほどの `reduce_mono` に加えて以下のような関数 `reduce1` を用意した．

```
Section reduce1.
Variables (A : Type) (op : A -> A -> A).
Definition reduce1 (a : assoc op) (d : A)
                   (x : list A) : A :=
  match x with
    | nil => a
    | a' :: x => foldl op a' x
  end.
End reduce1.
```

この `reduce1` は，第 1 引数として `assoc op` 型の項 `a` をとる．`assoc op` は，`op` が結合的な演算子であるという命題を表現した型である．`reduce1` は，第 2 引数のリストが `nil` の場合には定義されない部分関数と考えるのが自然であるため，4.2 節で説明した部分関数を定義する方針に従って実装されている．すなわち，第 2 引数が `nil` の時は，標準の値である `d` を返し，そうでない場合は，リストの先頭の値である `a'` を使い，`foldl` によって *reduce* を表現している．この `reduce1` も，`reduce_mono` と同じく，*reduce* の性質 (3) および (4) を満たしていることが証明できる．

以上のように，高階関数を定義する際に，関数だけではなく，関数がもつ性質の証明を引数として取ることで，より強力な運算規則を証明できる場合があることがわかる．

5 まとめと今後の課題

本稿では，プログラム運算を支援するために，Coq の tactic および記法のライブラリを開発したことを紹介した．また，Theory of Lists を表現する中で得られた，プログラムおよび証明を簡潔に，自然に記述するための手法について紹介した．

今後の課題として，自動的にプログラム運算をするための tactic や，ユーザーが自動プログラム運算を記述するのを支援するライブラリを実装することが考えられる．形式的に証明された定理のみを用いて，自動的にプログラム運算をするシステムがあれば，信頼性が高く，効果の高い効率化が行えると期待される．その点で，Ltac によって Coq に tactic を追加するという方法は，変更が容易で，自動証明をプログラミングでき，そのユーザーは対話的に使えるなどの性質をもつため，さまざまな応用の余地があると考えている．また，宣言的に証明を記述するための Coq の機能である C-zar の利用も議論してゆきたいと考えている．

謝辞 本稿の改善に協力した松崎公紀氏と森畑明昌氏に感謝する．

参 考 文 献

[1] The Coq Proof Assistant, `http://coq.inria.fr/`.

[2] Bertot, Y. and Castéran, P.: *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*, Texts in Theoretical Computer Science, Springer-Verlag, 2004.

[3] Bird, R. S.: An Introduction to the Theory of Lists. *Logic of Programming and Calculi of Discrete Design*, pp. 3–42. Springer-Verlag, 1987.

[4] Bove, A., Dybjer, P., and Norell, U.: A Brief Overview of Agda - A Functional Language with Dependent Types, *Theorem Proving in Higher Order*

Logics, 22th International Conference, TPHOLs 2001, Munich, German, August 17-20, Proceedings, Lecture Notes in Computer Science, Vol. 5674, Springer-Verlag, 2009. To appear.

[5] Chiba, Y., Aoto, T., and Toyama, Y.: Program Transformation by Templates based on Term Rewriting, *Proceedings of the 7th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 11-13 2005, Lisbon, Portugal*, ACM, 2005, pp. 59–69.

[6] de Moor, O. and Sittampalam, G.: Generic Program Transformation, *Third International Summer School on Advanced Functional Programming*, Lecture Notes in Computer Science, Vol. 1608, Springer-Verlag, 1999, pp. 116–149.

[7] Guttmann, W., Partsch, H., Schulte, W., and Vullinghs, T.: Tool Support for the Interactive Derivation of Formally Correct Functional Programs, *Journal of Universal Computer Science*, Vol. 9, No. 2, 2003, pp. 173–188.

[8] Hu, Z., Yokoyama T., and Takeichi M.: Program Optimizations and Transformations in Calculational Form. *Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2005), Braga, Portugal, 4-7 July, 2005*, Lecture Notes in Computer Science, Vol. 4143, Springer-Verlag, 2006, pp. 139–164.

[9] Mu, S.-C., Ko, H.-S., and Jansson, P.: Algebra of Programming using Dependent Types. *Mathematics of Program Construction, 9th International Conference, MPC 2008, Marseille, France, July 15-18, 2008, Proceedings*, Lecture Notes in Computer Science, Vol. 5133, Springer-Verlag, 2008, pp. 268–283.

[10] Smith, D. R.: KIDS: A Semiautomatic Program Development System, *IEEE Transactions on Software Engineering*, Vol. 16, No. 9(1990), pp. 1024–1043.

[11] Visser, E.: Stratego: A Language for Program Transformation based on Rewriting Strategies. System description of Stratego 0.5, *Rewriting Techniques and Applications (RTA'01)*, Lecture Notes in Computer Science, Vol. 2051, Springer-Verlag, 2001, pp. 357–361.