修士論文

定理証明支援系 Coq を用いたプログラム運算

48086225　**橋本 英樹**

指導教員　武市 正人 教授

2010 **年** 1 **月**

東京大学大学院情報理工学系研究科数理情報学専攻

概要

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．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開発に貢献すると期待される．しかし，支援システムがなければ正しい運算は難しいにもかかわらず，支援システム開発のコストの高さなどから，広く使われている支援システムは少ない．

本研究で，我々は定理証明支援系 Coq のライブラリによってプログラム運算を支援する手法を提案する。この手法は，Coq によって運算定理および運算定理の適用の正しさが保証される，対話的な運算ができる，拡張性が高いなどの特徴を持ち，その実装コストは高くない．

本論文では，プログラム運算支援について論じ，支援システムの実装となる tactic ライブラリを紹介する．また tactic ライブラリを用いたプログラム運算理論の表現および運算の実例を紹介し，その際に現れる諸問題と，対処法について論じる．

キーワード　プログラム運算, 定理証明支援系

目次

第 1 章

はじめに

本章では，本研究の背景となるプログラム運算および支援システムについて論じる．また，本研究の目的および貢献について説明する．

1.1 プログラム運算とは

プログラム運算 [3, 8] 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，その仕様から，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．

図 1.1 は，BMF (Bird-Meertens Formalism) [8] と呼ばれる理論に基づいてプログラムを書き換えている例である． 簡単な関数を関数結合で結合した形で書かれた，リストに含まれる最大部分列和を求めるプログラムが，プログラムの意味を変えない等式変形によって，計算量が改善されたものに変形されている．本節では，図 1.1 のようなプログラムの変形を可能にする理論について述べる．

本研究で述べるプログラム運算とは，BMF のようなプログラムを書き換えるための理論を用いて，プログラムを書き換えること全般を指すものとする．また，「運算」は，プログラムを書き換える過程を指すものとする．

BMF とは，等式による推論によって，プログラムを仕様から構成することを意図して集められた，プログラムを表現するための概念，記法，およびそれらの間に成り立つ定理 (以下，これらの定理を運算定理と呼ぶ．) の集まりである．全体として，プログラムを書き換えるための一つの理論をなしていると言える．

この，「プログラムを仕様から構成する」という考え方は，transformational programming [14] の考え方に基づいている．プログラマは最初から仕様を満たしかつ効率のよいプログラムを自分で書くのではなく，まずはなるべく簡潔で，間違いのない仕様をプログラムとして書くことに専念し，その後 BMF のような運算理論を用いてプログラムを書き換えればよい．これにより，仕様を満たしつつ，元々のプログラムが必要以上に複雑になることを避け，さらに運算理論によって効率のよいプログラムが導出されうるという特徴をもつ開発が行えると期待できる．

プログラムを効率化する，という観点から，仕様となるプログラムは，十分に抽象化された

$$
\begin{aligned}
& \underline{mss} \\
= & \{mss\text{の定義}, \uparrow \text{および} + \text{は結合的} \} \\
& \quad \uparrow / \circ (+/) * \circ \underline{segs} \qquad (\mathrm{O}(n^3)) \\
= & \{segs \text{ の定義} \} \\
& \quad \uparrow / \circ \underline{(+/) * \circ +\!\!+/} \circ tails* \circ inits \\
= & \{map \ promotion \text{則} \} \\
& \quad \underline{\uparrow / \circ +\!\!+/} \circ (+/) ** \circ tails* \circ inits \\
= & \{fold \ promotion \text{則} \} \\
& \quad \uparrow / \ \circ \ \underline{(\uparrow /) * \circ (+/) ** \circ tails*} \circ \ inits \\
= & \{map \text{分配則} \} \\
& \quad \uparrow / \circ (\underline{\uparrow / \circ (+/) * \circ tails}) * \circ inits \\
= & \{Horner \text{則}, (\uparrow, +) \text{は環} \} \\
& \quad \uparrow / \circ \underline{(\odot \not\rightarrow_0) * \circ inits} \\
= & \{scan \text{補題} \} \\
& \quad \uparrow / \circ (\odot \not\Rightarrow_0) \qquad (\mathrm{O}(n))
\end{aligned}
$$

ただし $x \odot y = (x + y) \uparrow 0$

図 1.1. BMF を用いたプログラム運算の例

高階関数を用いて記述されていることが望ましい．なぜなら，プログラムの一部の実行を最適化するだけでなく，アルゴリズムを記述するための高階関数自体を書き換えることで計算量のオーダーが改善する運算定理が知られているが，それら運算定理の適用可能性が容易に判断できるためである．

動的計画法 [6]，並列化 [10] などいくつかの問題領域では，それに特化したプログラム運算理論が提案されている．それらの理論が適用できる形で記述されているプログラムであれば，効率のよいプログラムに書き換えることができるといえる．

1.2　プログラム運算支援システムの必要性

プログラム運算によって仕様を満たすプログラムが開発できるためには，いくつかの条件が必要となる．

第一に，仕様となるプログラムが間違いなく書ける必要がある．そのためには，プログラムを書くための標準的な関数および結合子がライブラリとして整備され，文書によってライブラリの構成要素の意味が明確に定義されていることが望ましい．

第二に，仕様の段階で間違いがなくても，運算の過程で間違いを作りこんでしまっては意味がないので正しい運算定理が正しく適用される必要がある．運算定理がどのような前提のもとで成り立つのかが正確に記述され，また適用時にその前提が確認されねばならない．

第三に，運算定理を適用できる箇所を発見できなければならない．プログラムから運算定理を適用できる箇所を発見することは自明ではなく，人間の洞察により定理が適用可能な形に書き換えるか，または探索アルゴリズムが必要である．

第一の条件のためには，広く使われ，文書が豊富なプログラミング言語および，その言語の信頼できる処理系があれば十分である．いっぽう，第二，第三の条件のためには，支援システムが必要である．言語の意味を解釈できる処理系および，機械的にプログラムから運算定理が適用可能な箇所を探索し，書き換えるシステム (プログラミング言語がそのような機能を備えている場合もありうる)，また書き換えによってプログラムの意味が変わらないことを保証するために，検証，定理証明システムが必要になると考えられる．

1.3 本研究の目的と貢献

前小節で議論した，仕様を満たすためのプログラム運算の条件が満たされるような環境を構築すべく，さまざまなシステムがすでに考案され，実装されている．目的は違えど，プログラム運算，またはより広く transformational programming を支援するシステムは Kids [15], MAG [7]，Stratego [19]，Ultra [9]，Yicho [11]，RAPT [4] など多数存在する．

ただし，上記の条件を満たすシステムを構築する際，また言語の変化などに応じた保守の際に，大きな労力がかかるため，プログラム運算のための環境として広く使われているシステムはあまり多くない．

近年，Mu ら [13] は，依存型をもつプログラミング言語 Agda を用いて，証明を伴なうプログラム運算を記述するための AoPA ライブラリを提案した．このライブラリで記述された運算はそれ自体が証明になるように実装されているため，非常に信頼性が高い．また，Agda の機能を用いて運算定理の表現および証明，運算の正しさの検証が行われるため，運算支援を実現するための労力をさほど必要としない．しかし，Agda は自動証明を行う機能の拡張性が高くないため，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．

本研究の目的は，Mu らの研究を発展させ，よりプログラム運算に適した環境を構築することである．我々は，プログラム運算には，以下のような性質を持つことが望ましいと考えた．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．これは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く，運算にはしばしば人間の洞察を必要とするためである．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．

本研究の貢献は大きく 2 つある．第一に，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，等式による推論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．この tactic を用いた運算は，先述した，対話的な運算を行うことができる，運算がプログラムの意味を保存する，拡張性が高い，ソースコードの可読性が高い，などのよい性質を持つことを確認した．

システムとして Coq を選択した理由は，広く使われている定理証明支援系であり，tactic による対話証明が強力であり，さらに Ltac で拡張することができるためである．当初から意図していたわけではないが，型システムの表現力が非常に高いこと，記法を自由に再定義できること，などの特徴もまた，運算を自然に記述することに役立つことが分かった．

第二に，tactic 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] を取り上げ，我々のライブラリを用いて表現した．具体的には，関数を実装し，運算定理を証明し，プログラム運算を関数間の等式として証明した．これにより，tactic ライブラリがプログラム運算をより読みやすく，自然に記述することに役立つことを確認した．

また，Coq の制限や，本研究の目的との関係で直接的に表現できないレクチャーノートの記述を発見し，対処法について考察した．

1.4　本論文の構成

次章以降，本論文は以下のように構成される．2 章では，Coq の諸概念と機能を用いるためのコマンドについて述べる．3 章では，プログラム運算を Coq で表現する方針について論じる．4 章では，プログラム運算を記述するために我々が開発した tactic ライブラリを紹介する．5 章では，tactic ライブラリを用いて Coq で実際にプログラム運算を行った際に現れる諸問題と，いくつかの解決策について論じる．最後に，6 章で，まとめと今後の課題について述べる．

第 2 章

Coq の概要

Coq [1] 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．Coq は，表現力の高い型をもつラムダ計算である Calculus of Inductive Constructions と呼ばれる計算体系に基づいている [17] (以後，「項」は Coq の型システムにより拡張されたラムダ項を指すものとする)．型や関数を自ら定義することにより，関数型プログラミング言語として用いることができる． また，Curry-Howard 対応により，命題に対応づけられる型をもつ項を構成することにより，証明を構成することができる．Coq の型システムは依存型 [12] をもち，関数をはじめとした項をパラメータとして取る型を定義することにより，述語を表現することができる．さらに，それら述語を tactic と呼ばれる言語を用いて対話的に証明することが可能である．本章では，定義の与え方，対話証明モードでの証明の構成法など，本研究で重要と思われる諸概念と機能を呼び出すためのコマンドを簡単に説明する．詳細はリファレンスマニュアル [17] および教科書 [2] 等を参照されたい．

2.1 宣言と定義

Coq は他のプログラミング言語と同じように，変数の宣言と定義をする機能を備えている．宣言とは，識別子に対して型を割り当てることである．一方，定義とは，識別子に対して具体的な項を割り当てることである．宣言と定義は，ともにそれが有効な範囲をもつ．それはスコープとよばれる．スコープには，グローバルな (どこからでも参照できる) スコープと，ローカルな (プログラムの一部からのみ参照できる) スコープがある．宣言や定義を与えるコマンドは複数あるが，宣言や定義のスコープが異なるものがある．**Axiom** コマンドはグローバルな宣言を与える．一方，**Variable** や **Hypothesis** コマンドはローカルな宣言を与える．また，**Fixpoint** や **Definition** コマンドはグローバルな定義を与える．一方，**Let** コマンドはローカルな定義を与える[*1]．

[*1] その他，グローバルな宣言を与えるコマンドとして **Parameter** および **Conjecture** があり，グローバルな定義を与えるコマンドとして **Example** がある．

2.2　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．セクションは，ローカルな定義 (宣言) のスコープとなる．セクションの中でなされたローカルな宣言 (定義) は，セクション中の，その宣言 (定義) よりも後でのみ有効である．また，2 行目の **Variable** コマンドは，それを囲む最も内側のセクション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** コマンドで，新たな型を定義する．ここでは list を Type 型をもつ型[*2]として定義している．4 行目と 5 行目の “|” 以降が list の構成子で，それぞれ list 型をもつ nil, A →list →list 型をもつ cons であると宣言している．

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，セクションの中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，ローカルな宣言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∀(A : Type), list A に，それぞれ変化する．また，グローバルな定義がローカルな定義に依存している場合，ローカルな定義はグローバルな定義の中に展開される．

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] Coq では，型もまた型をもつ項である．

2.3 関数定義

関数を定義するには，**Fixpoint** コマンドを用いる[*3]．以下は，2 つのリストを引数にとり，それらを連結した新しいリストを返す関数 app を定義する例である．

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

2 行目の **Variable** コマンドはローカルな宣言である．3 行目の app が関数名であり，それに続く x y : list A は 2 つの仮引数 x および y と，その型が list A である事を表す．また，“{struct x} : list A” の部分が，x の構造が単調に減少することから再帰が停止すること，および結果の型が list A 型であることを表す．“:=” 以降が関数の定義で，その中では，**match** 構文によって，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 app 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 app 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 2.1 に，高階関数 foldr および foldl の Coq での定義を示す．これは，それぞれ以下の *foldr* および *foldl* のように計算を行う．

$$foldr\ (\oplus)\ e\ [a_1, a_2, \ldots, a_n] = a_1 \oplus (a_2 \oplus \cdots \oplus (a_n \oplus e))$$
$$foldl\ (\oplus)\ e\ [a_1, a_2, \ldots, a_n] = ((e \oplus a_1) \oplus a_2) \oplus \cdots \oplus a_n$$

定義した関数は，**Eval** コマンドを用いて簡約することができる．

```
Eval compute in
  ((fun x : list nat ⇒x ++ x) (1 :: 2 :: 3 :: nil)).
= 1 :: 2 :: 3 :: 1 :: 2 :: 3 :: nil
: list nat
```

compute は，値呼びによって項を簡約するという指示を与える．最後の 2 行が，Coq が出力する結果の項である．これにより，Coq の関数を計算に用いることができる．なお，上の例の (fun x : list nat ⇒x ++ x) は，x を引数に取って x ++ x を返すラムダ項である．また，“++” は app の中置演算子による表記である．

[*3] **Fixpoint** コマンドは再帰関数を定義するためのコマンドである．再帰的でない関数を定義する場合，**Definition** コマンドを用いる．

```
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 2.1. foldr および foldl の定義

2.4　命題の表現

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 [16] 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．また，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ればよい．

Coq の型システムは，依存型を含んでいる．ある型が値を引数に取ってはじめて型になる関数型として定義されているとき，その型を依存型 [12] であるという．たとえば，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
```

eq は本来 A : Type と x : A，さらに A と 3 つの引数を取り Prop[*4]に属する型を返す．しかし，構成子 refl_equal の型 eq x x は第 1 引数は省略されている．このように省略される引数を暗黙引数といい，**Set Implicits Arguments** コマンドにより新たに定義する関数の引数が可能なかぎり暗黙引数になる機能を用いている．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 **Definition** コマンドで確認することができる．

[*4] Prop は，Type と同じく，型が属する型である．

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 という定義を与え，それが 1 + 1 = 2 型をもつことを確認している．これにより，Prop 型に属する型 1 + 1 = 2 型をもつ項が構成されたので，すなわち 1 + 1 = 2 が証明されたことになる．oot は定理名として用いることができる．

2.5 対話証明モードと tactic

前節で，我々は **Definition** コマンドを用いて定義を与えることで証明を構成した．しかし，tactic を用いた証明が，より容易で広く使われる．**Theorem** コマンド[*5]で証明すべき命題 (型) を示すことで，tactic による証明を行う状態 (対話証明モード) になる．

今までユーザーが入力するコマンドのみ示してきたが，Coq は対話的に出力を返す．対話証明モードでは Coq からの出力をユーザーが確認することで，効率のよい証明が可能であるので，対話証明モードの出力を簡単に紹介する．

Coq < **Set Implicit Arguments**.

Coq < **Require Import** Plus.

Coq < **Section** tactic_demo.

Coq < **Theorem** thm_example : ∀l m n : nat, m = n →l + m = n + l.
1 subgoal

```
============================
 ∀l m n : nat, m = n →l + m = n + l
```

各行の Coq <の部分が Coq からのプロンプトであり，以降ピリオドまでがユーザーが入力したコマンドである．4 行目では，「任意の自然数 l, m, n について，m = n ならば，l + m = n + l である」を証明すべき定理 thm_example として **Theorem** コマンドを呼び出した．Coq は ∀l m n : nat, m = n →l + m = n + l を現在のゴールとして， ユーザーに tactic の入力を要求する．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．対話証明モードでは，証明すべき命題すなわち型がゴールとして示され，また現在のローカルなスコープに含まれる宣言および定義が前提として列挙される．

thm_example < intros.
1 subgoal

[*5] **Definition** コマンドで型のみを示し定義を示さないことで，**Theorem** コマンドと同じ効果をもつ．

```
  l : nat
  m : nat
  n : nat
  H : m = n
  ============================
   l + m = n + l
```

二重線の上側が現在の前提 (宣言および定義) を表し，下側が現在のサブゴール (ゴールを証明するために現在証明する必要がある命題を指す．複数ある場合もある) を表す． intros tactic によって，関数型だったゴールの，引数の型がすべて前提に移動したことがわかる．

```
thm_example < rewrite H.
1 subgoal

  l : nat
  m : nat
  n : nat
  H : m = n
  ============================
   l + n = n + l
```

rewrite tactic は，等式の証明を用いてゴールの一部を書き換える tactic である．この場合，rewrite の引数 H は m = n という等式の証明であり，ゴールの m が n に変化したことがわかる．

```
thm_example < apply plus_comm.
Proof completed.

thm_example < Qed.
intros.
rewrite H in ⊢∗.
apply plus_comm.

thm_example is defined

Coq < End tactic_demo.
```

apply tactic は，定理を適用することによって証明をする，という指示を与える tactic である．もし定理に前提があれば，その前提を次のゴールとして証明する必要があるが，この場合用いた定理 plus_comm には前提は必要ないので，これで証明は完了した．なお，plus_comm は，自然数の加算の交換法則を表す定理で，**Require Import** コマンドで読み込んだものを用いた．

証明が完了した状態で **Qed** コマンド[*6]を入力すると，対話証明モードで構成された証明が，thm_example の定義として環境に登録される．

その他，本論文に関係のある tactic として，eq の構成子である refl_equal を用いて等式を証明する reflexivity，不完全な項を与えて推論されない部分を次のサブゴールとする refine などがある．

また，Coq は，Ltac という，tactic をどのような順番で呼び出すかプログラムすることのできる言語を備えている．tactic の逐次適用，前提およびゴールの型による条件分岐，tactic が失敗した場合のバックトラックや例外処理など，基本的な tactic から複雑な tactic を構成するための機能が備わっている．

2.6 他言語システムとの連携

Extraction コマンドによって，Coq の項を他の言語の項として出力することができる．これにより，Coq で性質が証明された項 (プログラム) を，他言語の処理系で高速に実行することが可能である．また，Coq の項を，Coq では表現できないあるいは簡約できない項[*7]と共に計算に用いることも可能である．

現在, 対応している言語は OCaml, Haskell, Scheme である．たとえば, 関数 app を Haskell 言語で出力したい場合，以下のようなコマンドを入力すればよい．

```
Extraction Language Haskell.
Extraction "app.hs" app.
```

これにより，以下の内容をもつファイル App.hs が出力される．

```
module App where

import qualified Prelude

data List a = Nil
            | Cons a (List a)

app :: (List a1) -> (List a1) -> List a1
app l m =
  case l of
    Nil -> m
    Cons a l1 -> Cons a (app l1 m)
```

[*6] ゴールの型が Type に属する場合は，簡約の際に定義が展開される **Defined** コマンドを用いる方がよい．

[*7] **Fixpoint** コマンドで停止しない再帰関数を定義することはできない．また，**CoFixpoint** コマンドで定義された関数は簡約されない．

以上のように，関数 app および必要な定義 List が出力されていることがわかる．

第 3 章

Coq におけるプログラム運算の表現

本章では，Coq でプログラム運算を表現する方針について論じる．プログラム運算に限らず，Coq の表現力を活用してプログラミング言語を様々な方法で表現できることが知られている [5]．そこで，プログラム運算のためにはどのような表現を用いることができ，それぞれどのような利点や欠点があるかを論じる．

3.1 運算対象の構文領域と意味領域

まず，運算対象のプログラムをどのように表現するかについて論じる．一般に，プログラミング言語は，対象となる集合 (構文領域) のうち，意味領域と呼ばれる集合と対応づけられている元からなる部分集合である．Coq でプログラム運算を扱う際に，構文領域のみを扱い，運算定理として抽象構文木同士の関係のみを述語で与えるという方法も考えられる．しかしこの表現は，述語の定義に間違いがあっても確認する術がなく，安全なプログラム運算を行うという研究の目的から外れるので，本研究では扱わない．本研究の目的を鑑みると，Coq の表現力を活用して，意味領域を考慮するのが望ましい．そこで，まず構文領域と意味領域を設定する必要がある．

第一の方法は，Coq の関数を運算対象として扱うことである．この場合，構文領域は，Coq のすべての項であり，意味領域は，Coq での項の意味であるといえる．この方法は，様々な利点をもつ．たとえば，標準ライブラリに含まれている関数など，既存の関数を利用できる．また，関数そのものが意味であるため，引数を与えて **Eval** コマンドで簡約したり，**Extraction** コマンドによって他言語での計算に用いることもできる．また単一化の際に項が簡約されることは，後述の述語 eq を証明することを容易にする．

本研究では，以上のメリットを重視し，本論文の 5 章で扱う応用を，すべてこの方法によって記述した．この方法では，プログラムの型はたとえば∀A, list A →nat のように，項をとって項を返す関数型となる．

ただし，この方法は，Coq の関数に関する制限が強いという欠点をもつ．すなわち，関数はすべて，全域関数かつ停止する関数として記述しなければならない．この制限がプログラムの意味領域に対して強すぎる場合，他の方法をとる必要がある．

第二の方法は，構文領域を抽象構文木からなるデータ型として定義し，また意味領域を Coq の関数とする方法である．これは対象とする言語を Coq に埋め込む方法である．前述の Coq の関数のメリットを享受しつつ，対象言語を自由に設定することができるメリットがある．

第三の方法は，構文領域，意味領域ともにデータ型として定義する方法である．この方法は，意味領域で Coq の関数の制限にとらわれないというメリットがある．

第二の方法と第三の方法では，構文領域と意味領域の関係を与える必要がある．これには意味関数を与える方法と述語を与える方法がある．

意味関数を与える方法とは，構文木を引数にとって意味を返す Coq の関数を定義する方法で，プログラムの型をあらかじめ設定したデータ型 (たとえば Prog) とすると，第二の方法では，∀(p : Prog), typeDenote p のような型をもつ意味関数を与えることになろう．(Coq の項はすべて型をもつので，意味領域の型を計算する関数 typeDenote を与える必要がある．) 第三の方法では，意味領域の型をたとえば Sem とすると Prog →Sem 型の関数を与えることになるであろう．

一方，述語を与える方法とは，あるプログラムがある意味をもつ，という関係を **Inductive** コマンドで帰納的な述語として与える方法である．この方法では，必ずしもプログラムが意味をもつことが保証されないため，それを証明する必要がある．第三の方法では，Prog →Sem →Prop のような型をもつ述語 (たとえば interpret とする) を定義して，∀prog : Prog, exists sem : Sem, interpret prog sem のような interpret に関する定理を証明していく必要があろう．

3.2　運算定理の表現

次に，運算定理を表現する方針について論じる．最も単純な方法は，運算定理を Coq の項同士の等しさを表す述語 eq[*1] として表現する方法である．eq は，2 章で紹介したように実装されており，簡約で等しくなる項同士の等しさは型検査の際に自動的に確かめられる．eq は，Leibniz equality と同値であることが知られている [17].

ひとたび定理が等式として証明されれば，対話証明モードで等式の左辺 (右辺) がゴールの任意の場所に現れた際に rewrite tactic でそれを等式の右辺 (左辺) と置き換えることができる．一般に，関数型プログラミング言語によるプログラムは，一般に小さな関数を合成することで構成されることが多いが，小さな関数同士の等価性を多数証明しておくことで，より大きな関数の一部を書き換えることが可能になる．

この eq の問題として，外延的に等価な関数同士が等しいことが示せないという問題がある．たとえば，A →B 型の関数 f と g について，等式 f = g は f と g が構文上等しい関数でなければ示すことはできない．(引数を与えれば B 型の項同士を比較することはできる．) この問題を解決するには，後述の同値関係を用いるか，関数の外延的等価性の公理を導入する必要が

[*1] Coq の標準ライブラリには，同じとは限らない型をもつ項同士の等価性を表す述語 JMeq が用意されているが，本論文では扱わない．

ある．

関数の外延的等価性とは，定義域および値域が等しい 2 つの関数同士の関係で，それぞれが定義域全域で同じ値を返す関係のことである．外延的等価性の公理とは，外延的に等価な関数同士は等しいという公理である．Coq では標準ライブラリの Coq.Logic.FunctionalExtensionality モジュールに以下の型をもつ公理 functional_extensionality_dep として宣言されており，**Require** コマンドで読み込むことができる．

```
Axiom functional_extensionality_dep : ∀{A} {B : A →Type},
  ∀(f g : ∀x : A, B x),
  (∀ x, f x = g x) →f = g.
```

値域が入力によらず決まっている関数に対しては，この公理から導かれる定理 functional_extensionality が有用である．

```
Lemma functional_extensionality {A B} (f g : A →B) :
  (∀ x, f x = g x) →f = g.
```

この公理によって，∀x : A, f x = g x が証明されればただちに f = g を証明することができる．

一般的に公理を導入する欠点として，公理は単独で矛盾を生じることがなくても，複数の公理の組み合わせから矛盾を生じる場合があるので，現在どの公理が宣言されているかに留意する必要がある．

運算定理を表現するもう一つの方法は，関数の型を同値関係を伴なう型であるとみて関数同士は同値である，という述語によって二項関係を表現する方法である．たとえば，A →B 型の関数 f および g について，「任意の A 型の入力 a について，f と g は同じ値を返す」という命題 ∀x : A, f x = g x という関係を f および g について一般化すると，以下のように A →B 上の二項関係になる．ここで，relation (A →B) は (A →B) →(A →B) →Prop を表し，A →B 上の二項関係がこの型をもつ．

```
Section ext_eq.
Variable A B : Type.
Definition extensionally_eq : relation (A →B) :=
  fun f g : A →B ⇒∀x : A, f x = g x.
End ext_eq.
```

この関係が反射的，対称的，推移的であることは容易に証明でき，二項関係 eqtensionally_eq は同値関係である．

Coq では，その上に同値関係が定義された型を Add Parametric Relation コマンドによって登録すると reflexivity や rewrite などの tactic が拡張される機能が備わっており，ゴールが特定の型の場合に用いることができる．

eq は同値関係の特別な例であるため，同値関係を用いる方法がより一般的であるが，この方

法には rewrite による書き換えが eq と比べて制限されるという欠点がある．

運算定理の表現に，上記 2 つのいずれを用いたとしても運算定理の適用に前提が必要な場合がある．たとえば，A →B 型をもつ関数 f と g は，述語 P : A →Prop を満たすような a : A について全て同じ値を返す，という類の定理が考えられる．この定理は，∀a : A, P a →f a = g a と表現できる．この運算定理を用いてプログラムの書き換えを行うためには，P a の証明を構成しなければ ならない．このような前提を持つ運算定理は，前提のない運算定理のように直接書き換えに用いることはできないが，運算定理の適用条件を考慮に入れた運算が可能である．

3.3　運算の表現

続いて，運算を表現する方法について論じる．

基本的には，運算定理が eq 型で定義されていれば eq で，一方同値関係で定義されているならば同値関係で表現すればよい．これにより，プログラム運算自体が定理となり，信頼できる上に，より複雑な定理の証明に用いることができる．

また，別の方法として，運算でどのようなプログラムが結果として現れるか分からない場合に，sig を用いることができる．sig とは，subset type と呼ばれ，以下のように定義される型である．

```
Inductive sig (A : Type) (P : A →Prop) : Type :=
    exist : ∀x : A, P x →sig P.
```

適切な A および P について，sig P 型は，直観的には「述語 P をみたす A 型の項が存在する」という命題を表す[*2]．この sig 型に関して糖衣構文が用意されており，{x : A | P x} は sig (fun x : A ⇒P x) と同じ意味である．これを用いて，たとえば {g : A →B | f = g} によって「関数 f と等しい関数 g が存在する」という命題を表現することができる．構成子 exist に g として何か関数を与え (witness と呼ばれる)，f とその関数が等しいことの証明を与えることでその型をもつ項が構成される．この型の witness は，運算対象プログラムが Coq の関数そのものの場合，すでに定義された関数と同様に **Eval** コマンドや **Extraction** コマンドによって計算に用いることができる．

[*2] 「命題」には Prop に属する型である，という含意がある場合があるが，ここでは便宜上の言い方である．sig P は **Set** に属する型であるが Prop に属する型ではない．

第 4 章

プログラム運算記述のための tactic ライブラリ

2 章で述べた通り，tactic は対話的に証明を構成するための言語であり，Ltac によって拡張することができる．本章では，プログラム運算を自然に記述するために，我々が開発した tactic ライブラリについて紹介する．我々がプログラム運算支援システムとして意図した性質を述べ，また重要な tactic について述べる．各 tactic の詳細は，我々のテクニカルレポート [18] を参照されたい．なお，本 tactic ライブラリおよび 5 章の応用は，我々のプロジェクトページ (https://traclifo.univ-orleans.fr/SDPP/) から，上記テクニカルレポートのライブラリ (Library “Program Calculation in Coq”) として入手可能である．

4.1 Coq によるプログラム運算支援の方針

1 章で述べた通り，我々は対話的に安全な運算ができる，拡張性の高いプログラム運算支援システムを意図して開発を行った．

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，および運算定理を表現することができる．また，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，tactic で書かれた証明は読みづらく，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:
∀(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 ←IHl; auto.
Qed.
```

本研究では，Coq の利点を活用しながら，その欠点を補うことを意図して支援システムを開発した．

4.2　等式推論記述のための tactic ライブラリの機能

前節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 CalculationalFormProof 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．

4.2.1　運算状態の保持

CalculationalFormProof ライブラリでは，以下のような状態を表す型 state および state をパラメータに取る型 memo 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
| BOTH_SIDE : state
| ADHOC : ∀s:string, state.

Inductive memo (s: state) : Prop :=
  mem : memo s.
```

たとえば，現在等式がゴールで，その左辺を書き換えていることを表現したい場合には前提に memo LHS 型を持つ項を，右辺を書き換えていることを表現したい場合には前提に memo RHS 型を持つ項を追加できればよい．これは，以下の tactic で簡単に実現できる．

```
pose (mem _ : memo RHS)
```

前提に追加する項の型が重要な理由は，Ltac は前提に存在する項の型によって条件分岐を行うことができるためである．この性質をより有効に活用するために，我々は LHS t と RHS t の 2 つの tactic を用意した．LHS t は，tactic t を引数にとり，現在の前提に memo LHS 型の項を追加してから t を実行する．RHS t も同様である．ただし，両者とも，既に前提に memo s 型の項がある場合，それを消去してから追加を行う．これら tactic は，次小節の tactic と組み合わせて使うことを意図しており，左辺および右辺の書き換えが始まるタイミングが，ソース

コードから明確に分かるようになる．

4.2.2 等式変形による推論

次に，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
```

ただし，t および e は引数となる tactic および項である．この tactic は，前提に memo LHS 型の項がある場合，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
```

Ltac の **match** 構文は，前提とゴールの型によって適用する tactic を変えることのできる tactic である．全体として，「ゴールが等式のときに，その左辺で変数 lhs を束縛し，tactic t を実行することによって lhs = e を証明し h とし，rewrite h によってゴールを書き換える」という意味になる．この tactic を連ねて記述することで，等式変形のように運算を記述することができる．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 4.1 に示す．

4.2.3 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，その際の記述をより読みやすくするため，我々は既存の tactic にいくつかの別の表記を与えた．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，等式の証明を終了する際には， reflexivity の別表記として， “[]” が使えるようにした．

4.2.4 Subset Type の除去

3 章で議論したように，運算の結果が分かっていない場合には，sig 型が有用である．たとえば，ある関数 f : A →B を運算によって書き換えていきたい場合，証明すべき定理の型を {g : A →B | f = g} とすればよい．ただしこの場合，ゴールが等式ではないので，先述の ={ t } e tactic で書きかえることはできない。そこで次の定理 sig_elim のように，ゴールを等式にすることで，左辺のみを等式変形できる tactic を導入した．以下の Begin tactic がそれで

```
Lemma appAssoc:
∀(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 4.1. CalculationalFormProof ライブラリによるappAssocの証明

ある．

```
Definition sig_elim :
  {n : nat | 1 + 1 + 1 = n}.
Begin.
    LHS
={ by def plus }
    (2 + 1).
={ by def plus }
    3.
[].
```

Begin tactic は econstructor tactic を用いてゴールの右辺に対応する変数を用意する．それ以降はゴールが等式の場合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．

4.3 tactic ライブラリと自動証明 tactic との併用

本節では，前節の tactic ライブラリと自動証明 tactic との組み合わせについて述べる．前節で紹介したとおり，={ t } e tactic の第 1 引数 t は，現在のゴールの左辺 (右辺) が e と等しいことを証明するために使われる．この t として，複雑な定理が自動的に証明できる tactic を用いることで，人間が理解しやすい範囲で運算の 1 ステップを大きくすることができるため，より運算が読みやすくなることが期待できる．

4.3.1 ring および field

ring および field は，環および体の性質を用いて自動的に証明を構成する tactic である．環および体を新たに登録することで，交換法則，分配法則などを用いて自動的に証明を行う．たとえば，自然数が加算および乗算について環になっていることが証明され，登録されているとする．このとき，以下のように m ∗ n + m を (n + 1) ∗ m に書き換えることが可能である．これは，両者が等しいことが ring によって自動的に証明されるためである．

```
Section calc_ring.
Variables n m l : nat.
Theorem ring_example : {n' : nat | m * n + m = n'}.
Begin.
    LHS
={ ring }
    ((n + 1) * m).
[].
Qed.
End calc_ring.
```

ring および field は Coq の標準ライブラリに含まれるが，より一般的には，proof by reflection [2, 5] と呼ばれる手法により，ゴールの構造を反映した証明が Ltac で自動的に構成できることが知られており，運算に有用であると考えられる．

4.3.2 autorewrite

autorewrite は，rewrite タクティックと同様のゴールの書き換えを，繰り返し自動的に行う tactic である．複数の **Hint Rewrite** を含む HintDB (tactic から参照するための tactic の集まり) が与えられたときに，ゴールから等式の左辺に該当するパターンを発見し，等式の右辺に

書き換えることをパターンが発見できなくなるまで行う[*1].

使い方の例を述べる．関数合成で記述された関数を運算の対象にする場合，関数合成の結合性を使って結合の順序を変更したり，恒等関数 id を関数合成したい，あるいは関数合成の単位元として除去したい場合がある．これを便利にするために，以下のように，**Hint Rewrite** を用いて関数合成の結合性を表す compose_assoc と，恒等関数 id を左および右に合成しても関数はもとの関数と等しいことを表す compose_id_left および compose_id_right を，program_calculation と名づけられた HintDB に登録しておく．また，autorewrite を短い記述で呼び出せるように，program_simplify で autorewrite **with** program_calculation を呼び出すものとする．

```
Require Import Program.
Hint Rewrite ←compose_assoc : program_calculation.
Hint Rewrite compose_id_left : program_calculation.
Hint Rewrite compose_id_right : program_calculation.

Ltac program_simplify := autorewrite with program_calculation.
```

すると，以下のように，program_simplify で，id :o: (h :o: g) :o: f を h :o: (g :o: f) :o: id に書き換えることができる[*2]．これは，autorewrite により，両者が等しいものに書き換えられるためである．

```
Section autorewrite_example.
Variables A B C D : Type.
Variables (f : A →B) (g : B →C) (h : C →D).
Theorem autorewrite_test : {f' : A →D | id :o: (h :o: g) :o: f = f'}.
Begin.
    LHS
={ program_simplify }
    (h :o: (g :o: f) :o: id).
[].
Qed.
```

[*1] 等式を H とするとき，**Hint Rewrite** H で HintDB に追加した場合は左辺を右辺に書き換えるが，**Hint Rewrite** ←H で HintDB に追加した場合は右辺を左辺に書き換える．

[*2] 図 5.1 中の記号:o:は，関数合成演算子を表す．標準ライブラリの Coq.Program.Syntax モジュールには別の表記が用意されているが，本論文では可読性のため独自の記法を導入した．

第 5 章

Coq によるプログラム運算の実際

An Introduction to the Theory of Lists [3] (以下 Theory of Lists と呼ぶ．) 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，前章で紹介した tatic ライブラリで運算が自然に記述できることを確認するために，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，そして運算が証明としてレクチャーノートに近い形で表現できることを確かめた．そのコードは，およそ 4000 行である．

Theory of Lists を実装する中で，その定義や証明を自然に表現できない概念があり，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．本章の以下では，Coq での表現が自明ではないいくつかの概念について述べ，表現する方法について論じる．

5.1 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，高階関数に対して簡潔な記法が紹介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，運算が非常に読みやすく，理解しやすいものになっている．

Theory of Lists では，リスト上の高階関数の代表的なものとして，たと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．

$$f* \, [a_1, a_2, \ldots, a_n] = [f \ a_1, f \ a_2, \ldots, f \ a_n] \tag{5.1}$$

“/” は *reduce* を表し，結合的な二項演算を第 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.

$$\oplus/\ [a_1, a_2, \ldots, a_n] = a_1 \oplus a_2 \oplus \cdots \oplus a_n \tag{5.2}$$

reduce の第 2 引数のリストが空リストである場合の定義がいくつか考えられる．これについては本章の後半で論じる.

“◁” 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．Coq では，以下のように定義できる.

```
Section filter.
Variable A : Type.
Fixpoint filter (f : A →bool) (x : list A) : list A :=
  match x with
  | nil ⇒nil
  | a :: x' ⇒if f a then a :: filter f x' else filter f x'
  end.
End filter.
```

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，また，Bird の記法を参考に記法を定義した．その上で 4 章の tactic ライブラリを用いてプログラム運算を行った．図 5.1 に，CalculationalFormProof 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する． 図 5.1 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.

5.2　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．Coq で部分関数を表現する方法はいくつか考えられるが，それについて論じる.

第一に，関数の引数を 1 つ増やす方法が考えられる．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.
```

```
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.

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

□

ただし $f\ a = (p\ a)\ ?\ [a] : []$

図 5.1. 運算の例．(左) CalculationalFormProof ライブラリによるもの，(右) Bird の記法によるもの．

```
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，入力として本来未定義の nil の場合は d を出力としている．なお，この定義は，標準ライブラリの hd と同じものである．

この方法の利点は，部分関数に標準の値を与えた[*1]したものを全域関数と同様に扱うことができることである．head を例にとると，A 型の項 a について，head a の型は list A →A になり，他のリスト上の関数と同様の型をもつ．これにより，関数同士の合成は容易になる．

一方，この方法の欠点は，運算定理が複雑になりがちなことである．たとえば，「リストの先頭に要素を追加してから先頭を取ると，それは追加した要素になる」という定理を表現することを考える．もし head が全域関数であれば，以下の型で表現できるであろう．

[*1] カリー化された関数には，引数のすべてを与えるするだけでなく一部の引数のみ渡すことができる．一部の引数のみ渡すことを部分適用という．

```
Theorem head_cons :
  ∀(a : A) (x : list A), head (cons a x) = a.
```

しかし，head は部分関数なので，以下のように引数として標準の値を取らねばならない．

```
Theorem head_cons :
 ∀(a d : A) (x : list A), head d (cons a x) = a.
```

このように，得られた定理は標準の値 d に関して一般化されており，定理の意味を理解しづらくなっている．

第二に，戻り値の型が A のときに，それを option A 型にする方法が考えられる．option 型とは，A 型の値をもつ構成子 Some と，値をもたない構成子 None から構成される型であり，以下のように定義されている．

```
Inductive option (A : Type) : Type :=
  | Some : A →option A
  | None : option A
```

これを用いて，先の関数は以下のように表現できる．

```
Definition head_option (x : list A) : option A :=
  match x with
    | nil ⇒None
    | a :: x' ⇒Some a
  end.
```

option 型によって，パターンマッチに失敗したという「例外」[*2]を表現する手法は Coq に限らず関数型言語でしばしば使われる手法である．この方法の利点は，関数が None を返したときに，パターンマッチが失敗したことが分かる点である．第一の方法では，標準の値が返ってくるので，パターンマッチに失敗したのか，または成功して関数が標準の値と同じ値を返したのかどうかが区別できない場合がある．一方，この方法の欠点として，option 型を返す関数と返さない関数が混在している際に option 型を返さない関数を option 型を返す関数に変換する[*3]必要があり，この変換に関する新たな運算定理が必要になるため，運算が冗長になると考えられる．なお，head_option の定義は標準ライブラリの head と同じ定義である．

第三に，関数の引数に続いて，引数に関する述語の証明を引数にとる方法が考えられる．この方法では，先の関数は以下のように表現できる．

[*2] 一般に，プログラミング言語での例外は，∀A : Type, A 型をもつが，このような型は Coq では矛盾を導くため，宣言されるべきではない．ここでは便宜上の言い方である．

[*3] この変換は，リフトと呼ばれる．

```
Definition head_premise (x : list A) : x <> nil →A :=
  match x return x <> nil →A with
    | nil ⇒fun prf ⇒match prf (refl_equal nil) with end
    | a :: x' ⇒fun _ ⇒a
  end.
```

ここでは，引数であるリスト x に加えて，「x は nil と等しくない」を表す x <> nil を引数に取っている．そして，x についてパターンマッチを行い，x が nil のとき，x <> nil を用いて矛盾を導き[*4]，A 型の項を返す関数を記述している． この方法で部分関数を実装すると，関数同士を関数合成によって自在に結合することができなくなってしまうため，関数の一部を運算定理によって書き変えていく，という目的にはそぐわないといえる．

第四に，関数の定義域を表現した新たな型を定義することが考えられる．この方法では，先の関数は新たに定義する型 list1 を用いて以下のように表現できる．

```
Section list1.
Variable A : Type.
Inductive list1 : Type :=
  | single1 : A →list1
  | cons1 : A →list1 →list1.

Definition head_on_list1 (x : list1) :=
  match x with
    | single1 a ⇒a
    | cons1 a x' ⇒a
  end.
End list1.
```

この方法を採用すると，関数の定義域に応じて型を定義する必要があるため，第三の方法と同じく，関数合成が自由にできなくなってしまうと考えられる．

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，統一するのが望ましい．我々は，第一の方法，すなわち部分関数に 1 つ引数を追加する方法の多くの利点を重視し，これを採用した．

5.3 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 章で示した，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

[*4] x <> nil は (x = nil) →False を表す．False は構成子を持たない型で矛盾を表現し，ここでは prf (refl_equal nil) の型が False になる．

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．たとえば 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 リストは，Coq で以下の型として定義することができる．

```
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.
```

ただし，このような型を定義してしまうと list 上の関数，slist 上の関数，jlist リスト上の関数が混在し，関数同士の結合が難しくなってしまう．そこで，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 リスト上の定義を反映した定理を証明した．

まず帰納法の原理について述べる．2 章で **Inductive** コマンドを用いて list を定義したが，Coq ではその際に list 型に関する帰納法の原理が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．

```
list_ind :
  ∀(A : Type) (P : list A →Prop),
  P nil →
  (∀ (a : A) (l : list A),
   P l →P (a :: l)) →
```

```
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 5.2. snoc を利用した foldl の定義

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

図 5.3. `foldl_rev_char` の型

```
∀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．その代わりに，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 :
  ∀(A : Type) (P : list A →Prop),
  P nil →
  (∀ (x : A) (l : list A),
   P l →P (l ++ [x])) →
  ∀l : list A, P l

join_induction :
  ∀(A : Type) (p : list A →Prop),
  p nil →
  (∀ a : A, p ([a])) →
  (∀ x y : list A,
   p x ∧p y →p (x ++ y)) →
  ∀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 章で，*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，

先述の slist の定義を用いて図 5.2 のように定義できる．しかし 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 5.3 に示される型を持つ定理 foldl_rev_char を証明した．なお，x ++ [a] は，app 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 章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．この定理はまた，2 つの性質を満たす関数が一意に定まることを表しており，上の foldl_snoc 関数の定義を反映しているといえる．参考までに，foldl_rev_char を証明する手順の例を付録 A に示す．

5.4　二項演算の性質を利用する方法

5.1 節で紹介した高階関数 *reduce* は，以下の性質をもつ．

$$\forall a,\ \oplus/[a] = a \tag{5.3}$$

$$\forall x\ y,\ x \neq nil \wedge y \neq nil \Rightarrow \oplus/(x +\!\!+ y) = (\oplus/x) \oplus (\oplus/y) \tag{5.4}$$

直観的には，性質 (5.4) 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 では証明が項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．(monoid の定義は，付録 B に示す．) そして，この関数は結果として，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，

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

となるからである．同様に，この定義によって，二項演算の性質より *reduce* の性質 (5.3) および (5.4) が満たされる (実際に証明することができた．)．特に，この定義では x または y が *nil* の場合でも (5.4) 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 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 ⇒d
    | 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，5.1 節で説明した，標準の値を用いて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' を用いて，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 (5.3) および (5.4) 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．

5.5 Fictitious Value を用いた *reduce* の表現

本章の以降では，我々の実装でうまく表現できなかった概念を紹介していく．

まず，本節では，fictitious value [3] を用いて *reduce* を表現する際に問題が生じたことを説明する．Fictitious value とは，関数が未定義の場合の仮の値である．Theory of Lists には，前節で説明した *reduce* の第 1 引数である関数が結合的であるが，単位元をもたない場合に，前節で紹介した reduce1 を用いるのではなく，fictitious value を用いる方法が紹介されている．まず，二項演算子 $\oplus$ を，集合 A 上の二項演算とする．次に，e を，A に含まれるすべての値と区別される特別な値とする．そして，新たな演算子 $\oplus'$ を，以下のように定義する．

$$\begin{aligned} a \oplus' b &= b, \quad \text{if } a = e \\ &= a, \quad \text{if } b = e \\ &= a \oplus b, \quad \text{otherwise} \end{aligned}$$

これにより，二項演算子 $\oplus'$ は単位元 e をもつ．すなわち $(A, \oplus', e)$ はモノイドをなす．その上で，*reduce* は第 2 関数のリストが空リストの場合，e を返すものと定義すれば，*reduce* $\oplus'$ は前節で述べた reduce_mono と同様によい性質をもつ．

しかし，これを Coq で実装する場合，問題がある．その実装には，2 つの方針が考えられる．第 1 の方針は，e が A に含まれるとして宣言してしまうことである．しかし，$\oplus'$ を計算に用いるためには A 型の値と e が等しいかどうか決定される必要がある．宣言した e は定義を持たないので，そのような決定はされず，$\oplus'$ を計算に用いることができなくなってしまう．これは，Coq の関数を運算対象にしたことで，運算対象が計算に用いることができる，という利点を損なうものである．

第 2 の方針は，e を $A \cup \{e\}$ という新たな型に属するものとし，$\oplus'$ を $A \cup \{e\}$ 上の二項演算として定義する．$A \cup \{e\}$ は，A を Coq の A 型とみなした場合，option A と同型であり，実際のところ 5.2 節で論じた，部分関数の表現に option 型を用いた場合に対応する．5.2 節で論じた通り，この表現は他の関数と結合する際に記述量を増やすという欠点を持つため，我々はこの実装を採用しなかった．参考までに Coq で実装したものを付録 C に示す．

第 6 章

おわりに

6.1 まとめ

本研究では，いくつかの良い性質を持つプログラム運算システムを意図して開発を行った．その性質とは，運算によって意味が保存されることが保証されるという意味で安全であること，対話的に運算が行えて人間の直観を活かせること，拡張性が高いこと，運算の過程が理解しやすい形で保存されることである．

我々は，Coq の tactic を拡張する Ltac によって，上記の性質をもつシステムが Coq 上に容易に構築できることを確認した．具体的には，プログラム運算自体が証明になるため上記の意味で安全である．ユーザーは tactic によって逐次的に書き換えを行い，その結果を確認することができる．運算のための tactic である=`{ t } e` は，任意の tactic と組み合わせて用いることができるために定理や tactic の追加で拡張することができる．ソースコードは等式による推論を自然に記述でき，可読性は高い．また，当初から意図していたわけではないが，型システムの表現力が高く定理とプログラムを同じ言語で表現でき，*reduce* のようなよい性質をもつ高階関数を，二項演算の性質を利用して表現することができることがわかった．

さらに，tactic ライブラリの応用例として，Theory of Lists を表現することで，tactic ライブラリの有用性を確認した．等式の推論による運算の記述に加え，Bird の記法を用いることで，より簡潔に運算が記述できることを確認した．またプログラムおよび証明を記述する際に，直接的に表現できない概念に注目し，それに対処するいくつかの手法について考察した．

6.2 今後の課題

今後の課題を 3 点述べる．第一の課題として，Ltac によって運算の 1 ステップを大きなものにすることが考えられる．等式による推論は，ソースコードの可読性を高める効果があるが，その 1 ステップが小さすぎる場合，ソースコードが冗長になる可能性がある．そこで，運算の 1 ステップを，理解しやすい範囲でより大きな書き換えにできることが望ましい．そのために，運算に頻繁に現れるパターンを発見し，定理とするか，または自動的に証明がなされるような tactic を用意することで，より運算が理解しやすくなることが期待される．例として，

4 章で ring および field と，autorewrite を挙げたが，より複雑な tactic の使用も考えられる．

第二の課題として，1 章で，プログラム運算が特定の問題領域においてプログラムの効率化に貢献していることを述べた．このような既に知られている効率化はしばしば自動的に行うことが可能であり，プログラムの開発を容易にすると期待される．そこで，この種の効率化を Coq の上で tactic によって自動的に行うことが考えられる．これにより，プログラムの自動効率化が信頼できるものになると考えられるが，運算定理が適用できる箇所を tactic で発見できるかについて，考察と実験が必要である．

第三の課題として，様々な方法でプログラミング言語を実装し，プログラム運算がうまく表現できるのかを確認することが考えられる．本研究では 5 章で示した方法以外に実用的な規模の実装を行っていない．本論文の 3 章では，プログラム運算を Coq で表現する方法はいくつもの可能性があり，それぞれ利点や欠点をもつことを述べた．また，5 章では，部分関数や *reduce* の実装にいくつかの方法があることを述べた．それらの表現方法に対して，4 章で紹介した tactic はうまく動作するのか，また 5 章で紹介した問題への対処法がうまく機能するのかどうかについては，より多くの例が必要であるといえる．

謝辞

本論文の執筆にあたり，たくさんの人のお世話になりました．

指導教員の武市先生には，論文執筆，発表，研究の計画など研究の進め方をご指導いただきました．ありがとうございました．

国立情報学研究所の胡振江教授には，論文執筆，発表，ライブラリの実装方針など多面からご指導をいただきました．ありがとうございました．

共同研究者の Orleáns 大学の Frédéric Loulergue 教授，Julien Tesson 氏には，ライブラリの重要な部分を作っていただき，また Coq の様々な機能やテクニックを教えていただきました．ありがとうございました．

穆信成特任准教授には，依存型プログラミングの表現力など，テクニカルなアドバイスをいただきました．ありがとうございました．

松崎公紀高知工科大学准教授，江本健斗助教，森畑明昌氏，松田一孝氏には構成，文献の探し方，LaTeX の使い方など，論文執筆を基礎からを教えていただきました．ありがとうございました．

参考文献

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

[2] Yves Bertot and Pierre Casteran. *Interactive theorem proving and program development – Coq'art: The calculus of inductive constructions*. Springer-Verlag, 2004.

[3] Richard Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pp. 5–42. Springer-Verlag, 1987.

[4] Yuki Chiba, Takahito Aoto, and Yoshihito Toyama. Program transformation by templates based on term rewriting. In *PPDP '05: Proceedings of the 7th ACM SIGPLAN international conference on Principles and Practice of Declarative Programming*, pp. 59–69, New York, NY, USA, 2005. ACM.

[5] Adam Chlipala. Certified programming with dependent types. http://adam.chlipala.net/cpdt/.

[6] Oege de Moor. *Categories, relations and dynamic programming*. Ph.D thesis, Programming research group, Oxford Univ., 1992. Technical Monograph PRG-98.

[7] Oege de Moor and Ganesh Sittampalam. Generic program transformation. In *Proc. 3rd International Summer School on Advanced Functional Programming*, Vol. 1608 of *Lecture Notes in Computer Science*, pp. 116–149. Springer-Verlag, 1998.

[8] Jeremy Gibbons. An introduction to the Bird-Meertens formalism. In Steve Reeves, editor, *Proceedings of the First New Zealand Formal Program Development Colloquium*, pp. 1–12, November 1994.

[9] Walter Guttmann, Helmuth Partsch, Wolfram Schulte, and Ton Vullinghs. Tool support for the interactive derivation of formally correct functional programs. *J. UCS*, Vol. 9, No. 2, pp. 173–188, 2003.

[10] Zhenjiang Hu, Masato Takeichi, and Wei-Ngan Chin. Parallelization in calculational forms. In *25th ACM Symposium on Principles of Programming Languages (POPL'98)*, pp. 316–328, San Diego, California, USA, January 1998. ACM Press.

[11] Zhenjiang Hu, Tetsuo Yokoyama, and Masato Takeichi. Program optimizations and transformations in calculation form. In Ralf Lämmel, João Saraiva, and Joost Visser, editors, *GTTSE*, Vol. 4143 of *Lecture Notes in Computer Science*, pp. 144–168. Springer, 2006.

[12] James McKinna. Why dependent types matter. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *POPL*. ACM, 2006.

[13] Shin-Cheng Mu, Hsiang-shang Ko, and Patrik Jansson. Algebra of programming in Agda: Dependent types for relational program derivation. *J. Funct. Program.*, Vol. 19, No. 5, pp. 545–579, 2009.

[14] Robert Paige. Transformational programming: Applications to algorithms and systems. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 73–87, New York, NY, USA, 1983. ACM.

[15] Douglas R. Smith. Kids: A semi-automatic program development system. *IEEE Trans. Software Eng.*, Vol. 16, pp. 1024–1043, 1990.

[16] Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard isomorphism*, Vol. 149 of *Studies in Logic and the Foundations of Mathematics*. Elsevier Science Inc., New York, NY, USA, 2006.

[17] The Coq Development Team. The Coq Proof Assistant Reference Manual. http://coq.inria.fr/refman/.

[18] Julien Tesson, Hideki Hashimoto, Zhenjiang Hu, Frédéric Loulergue, and Masato Takeichi. Program calculation in Coq. Technical report, LIFO, University of Orléans, 2009.

[19] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies - system description of stratego 0.5. In *Rewriting Techniques and Applications (RTA' 01)*, Vol. 2051 of *Lecture Notes in Computer Science*, pp. 357–361. Springer-Verlag, 2001.

付録 A

異なるデータ型上の定義を利用する方法

5 章で，*foldl* の snoc リストの定義を反映した定理である foldl_rev_char について説明した．ここでは，その定理を証明する手順の例を紹介することで，なぜそのような定理が現れたかについて述べる．

まず，*foldl* が満たすべき仕様を記述する．snoc リスト上の *foldl* の定義である 5 章の図 5.2 は，cons リストでは，引数のリストが nil である場合および snoc a x (cons リスト上では x ++ [a] で表現できる) である場合に満たすべき性質として記述できる．そこで， 関数の実装と，それぞれの場合に満たすべき性質を抽象化し，foldl_impl，foldl_nil_case および foldl_snoc_case と名前を付けておく．この仕様をレコードとして，foldl_spec と名付ける．なお，レコードとは，構成子をただ 1 つのみもつ型で，フィールドと呼ばれる射影関数を伴なう．この場合，foldl_spec 型は構成子 Build_foldl_spec のみから構成される型で，foldl_impl は foldl_spec を引数にとり，list A →B 型の関数を取り出す射影関数である．

次に, 仕様 foldl_spec を構成する実装 foldl_impl は, 全て等しいことを証明する．これは, snoc リスト上の帰納法の原理である rev_ind を用いて容易に証明できる．この事実を，foldl_unique とする．これにより，foldl_spec が構成できれば，foldl_spec は well-defined な定義を与えているといえる．

続いて，cons リスト上に定義された関数 foldl を，foldl_spec の foldl_impl として，foldl_spec 型の項を構成する．そのためには，foldl_nil_case および foldl_snoc_case の型に現れる foldl_impl を foldl で置き換えた定理を示し，foldl_spec の構成子 Build_foldl_spec に与えればよい．(ただしここでは，foldl_snoc_case の証明は省略し，仮定 (Hypothesis) とした．) 構成された foldl_spec 型の項に，foldl_instance と名付けておく．

最後に，foldl_unique および foldl_instance を用いて，foldl_spec の性質 foldl_nil_case および foldl_snoc_case を満たすような関数の実装は，foldl に等しいと証明することができる．これにより，foldl が foldl_spec を構成する唯一の foldl_impl であることが表現される．

全体のソースコードを以下に示す.

Require Import List FunctionalExtensionality Program.

```
Set Implicit Arguments.

(** The specification of foldl. foldl is a function that satisfy
  foldl_nil_case and foldl_snoc_case *)
Record foldl_spec {A B : Type} (op : B →A →B) (e : B) := {
  foldl_impl : list A →B;
  foldl_nil_case : foldl_impl nil = e;
  foldl_snoc_case : ∀(x : list A) (a : A),
    foldl_impl (x ++ [a]) = op (foldl_impl x) a
}.

(** Two "foldl_impl"s are (extensionally) equal,
  if they both satisfy foldl_nil_case and foldl_snoc_case *)
Theorem foldl_unique : ∀(A B : Type) (op : B →A →B) (e : B)
  (f1 f2 : foldl_spec op e),
    foldl_impl f1 = foldl_impl f2.
intros.
destruct f1.
destruct f2.
unfold foldl_impl.
apply functional_extensionality.
apply rev_ind.
rewrite foldl_nil_case0; rewrite foldl_nil_case1; reflexivity.
intros.
rewrite foldl_snoc_case0; rewrite foldl_snoc_case1.
rewrite H; reflexivity.
Qed.

(** An implementation of foldl. *)
Fixpoint foldl {A B : Type} (op : B →A →B) (e : B) (x : list A) : B :=
  match x with
    | nil ⇒e
    | a :: x' ⇒foldl op (op e a) x'
  end.

(** A hypothesis that the "foldl" satisfies foldl_snoc_case.
  Actually, it can be shown *)
Hypothesis foldl_snoc_conc :
  ∀(A B : Type) (op : B →A →B) (e : B) (a : A) (x : list A),
    foldl op e (x ++ [a]) = op (foldl op e x) a.

(** The "foldl" makes a foldl_spec *)
Definition foldl_instance {A B : Type}(op : B →A →B) (e : B) :
```

```
  foldl_spec op e :=
    let foldl_impl := foldl op e in
    let foldl_nil_case := refl_equal e in
    let foldl_snoc_case := fun x a ⇒foldl_snoc_conc op e a x in
      Build_foldl_spec op foldl_impl foldl_nil_case foldl_snoc_case.

(** If a function f satisfy foldl_nil_case and foldl_snoc_case,
 it is equal to (foldl op e) *)
Theorem foldl_rev_char : ∀{A B : Type}(op : B →A →B) (e : B)
  (f : list A →B) ,
    f nil = e →
    (∀ (x : list A) (a : A), f (x ++ [a]) = op (f x) a) →
    f = foldl op e.
intros A B op e f H_nil H_snoc.
change (let foldl_instance_f := (Build_foldl_spec op f H_nil H_snoc) in
  foldl_impl foldl_instance_f =
    foldl_impl (foldl_instance op e)).
apply foldl_unique.
Qed.
```

付録 B

二項演算の代数的性質を記述する方法

5 章で，我々は二項演算がモノイドである (あるいは，結合的である) ことの証明を引数にとることによって，高階関数 *reduce* がよい性質をもつことを述べた．ここに，monoid および assoc の定義を示す．

monoid は，assoc および has_unit との連言によって定義されている．対話証明モードで前提に monoid op e がある場合，その連言を destruct などの tactic を用いて分解することができる．これを推し進めて，monoid_expand のように，Ltac によってその分解の仕方をプログラムするのが便利である．

また，monoid_left_unit および monoid_right_unit のように，monoid op e から直ちに導けるような，運算に用いることのできる代数的性質がある場合は，定理として証明しておくとよい．

```
Set Implicit Arguments.
Section Monoid.
(** op is associative. *)
Definition assoc (A : Type) (op : A →A →A) :=
  ∀(x y z : A), op (op x y) z = op x (op y z).

(** e is a identity unit of op *)
Definition has_unit (A : Type) (op : A →A →A) (e : A) :=
  ∀(x : A), op x e = x ∧op e x = x.

(** A is a monoid with respect to operator op and element e *)
Definition monoid (A : Type) (op : A →A →A) (e : A) :=
  assoc op ∧has_unit op e.

(** This tactic destruct monoid structure. *)
Ltac monoid_expand H :=
  unfold monoid in H; elim H; intros Assoc Unit; unfold has_unit in Unit.
```

```
(** This extract left unit law of the monoid. *)
Lemma monoid_left_unit:
  ∀(A : Type) (op : A →A →A) (e : A),
  monoid op e →
  ∀(x : A), op e x = x.
Proof.
intros A op e mono.
monoid_expand mono.
exact (fun x ⇒proj2 (Unit x)).
Qed.

(** This extract right unit law of the monoid. *)
Lemma monoid_right_unit:
  ∀(A : Type) (op : A→A→A) (e : A),
  monoid op e →
  ∀(x : A), op x e = x.
Proof.
intros A op e mono.
monoid_expand mono.
exact (fun x ⇒proj1 (Unit x)).
Qed.

(** monoid_assoc extract associativity from monoid structure. *)
Lemma monoid_assoc :
  ∀(A : Type) (op : A →A →A) (e : A),
  monoid op e →assoc op.
Proof.
intros.
monoid_expand H.
apply Assoc.
Qed.

End Monoid.
(** monoid_expand tactic destruct monoid structure. *)
Ltac monoid_expand H :=
  unfold monoid in H; elim H; intros Assoc Unit; unfold has_unit in Unit.
```

付録 C

option 型を用いた fictitious value の表現

5 章で，我々は option 型を用いることで fictitious value を用いた *reduce* が表現できることを述べた． ここに，実装を示す．

```
(** * Fictitious values can be implemented using option type. *)

Require Import List Plus.
Set Implicit Arguments.

Section reduce_option.
Variable A : Type.
Variable op : A →A →A.
(** Definition of associativity *)
Definition assoc := ∀a b c : A, op (op a b) c = op a (op b c).

Variable op_assoc : assoc.
(** lift2 lifts a binary operator *)
Definition lift2 (oa1 : option A) (oa2 : option A) :=
  match oa1, oa2 with
    | None, _ ⇒oa2
    | _, None ⇒oa1
    | Some a1, Some a2 ⇒Some (op a1 a2)
  end.

(** op' denotes the lifted function *)
Notation "'op'" := lift2.

(** If op is associative, then lifted function is also associative *)
Lemma lift2_assoc : ∀oa ob oc : option A,
  op' (op' oa ob) oc = op' oa (op' ob oc).
```

```
intros.
case oa; case ob; case oc;
intros; simpl;
try rewrite op_assoc; reflexivity.
Qed.

(** If the list is nil, reduce is not defined.
  If it is not, it is defined using op' *)
Fixpoint reduce_option (x : list A) : option A :=
  match x with
    | nil ⇒None
    | a :: x' ⇒op' (Some a) (reduce_option x')
  end.

(** It is also possible to implement reduce_option
  using foldr or foldl *)
Definition reduce_option_foldr : list A →option A :=
  fold_right (fun a : A ⇒op' (Some a)) None.

Notation "'red'" := reduce_option.

(** reduce and singleton *)
Theorem reduce_singleton' : ∀a : A,
  red (a :: nil) = Some a.
reflexivity.
Qed.

(** reduce and app *)
Theorem reduce_app' : ∀x y : list A,
  red (x ++ y) = op' (red x) (red y).
induction x.
intros.
simpl; trivial.
intros.
change (op' (Some a) (red (x ++ y)) = op' (op' (Some a) (red x)) (red y)).
rewrite IHx.
rewrite lift2_assoc.
reflexivity.
Qed.

End reduce_option.

(* examples. *)
```

```
(*
Eval compute in reduce_option plus nil.
Eval compute in reduce_option plus (1 :: 2 :: 3 :: nil).
Eval compute in reduce_option minus (10 :: 5 :: 1 :: nil).
*)

Section reduce.
Variable A : Type.
Variable op : A →A →A.
(** reduce takes only associative operators *)
Definition reduce (op_assoc : assoc op) :=
  reduce_option op.

Variable op_assoc : assoc op.
Theorem reduce_singleton :
  ∀a : A,
    reduce (op_assoc) (a :: nil) = Some a.
apply reduce_singleton'.
Qed.

(** reduce always distributes over app *)
Theorem reduce_app :
  ∀x y : list A,
  reduce (op_assoc) (x ++ y) =
    lift2 op (reduce op_assoc x) (reduce op_assoc y).
intros.
apply reduce_app'.
assumption.
Qed.
End reduce.

Lemma plus_assoc : assoc plus.
unfold assoc; intros.
apply plus_assoc_reverse.
Qed.

(* same as the last examples, but reduce is restricted *)
(*
Eval compute in reduce plus_assoc nil.
Eval compute in reduce plus_assoc (1 :: 2 :: 3 :: nil).
*)
```