# 高度な運算定理の Coq による証明とその自動化

村田 康佑　江本 健斗

プログラム運算とは，プログラムの代数的変形によって，素朴なプログラムから効率的なプログラムを得る手法である．Tesson らは，Coq を用いてプログラム運算の正しさを検証する手法を提案し，Bird によるレクチャノートである Theory of List に現れる運算規則の証明を形式化した．我々は，この手法をベースにして，任意の代数データ型に対して適用可能な再帰図式に関する運算規則を証明し，また Coq プログラムに定理を適用して高速な Coq プログラムが得られることを示した．その核は，始代数や終余代数を型クラスを用いて表現することにあり，インスタンス化することによって，実行可能な Coq プログラムに運算規則を適用することができるようになる．本発表では，そのインスタンス化において要求される証明の一部を自動化する手法を提案する．

## 1　はじめに

素朴なプログラムは実装しやすい一方，効率が悪くなってしまいがちである．対照的に，効率の良いプログラムは実装が難しいことが多い．一見して二律背反に思える二つの性質「プログラムの効率の良さ」と「実装のしやすさ」を両立することは，プログラミングの研究における重大なテーマである．

プログラム運算は，単純なプログラムから代数的変形を繰り返すことで，効率の良いプログラムを導出する技術である．変形にあたっては，次のマップ融合則のような変換規則を用いる:

$$\mathsf{map}\ g \circ \mathsf{map}\ f = \mathsf{map}\ (g \circ f).$$

ここで，map は，与えられた関数を，与えられたリストの各要素に適用する高階関数である: $\mathsf{map}\ f\ [a_1, \ldots, a_n] = [f\ a_1, \ldots, f\ a_n]$．図 1 (a) は，マップ融合則の直観的な説明である．

本研究の究極の目的は，プログラム運算の「正しさ」を保証するための安価な仕組みを構築することにある．この「正しさ」は，次の二つの側面から成る:

1. 変換規則自体の正しさ，
2. 変換規則の使い方の正しさ．

これらの証明は機械的に検証されるのが望ましい．

機械的検証をするシステムを構築する上で，対話的定理証明支援系 Coq [11] は良い選択である．実際，Tesson ら [10] は，プログラム運算のための Coq タクティックライブラリを構築し，その上でプログラム運算の教科書である Theory of Lists [2] に現れる運算法則の検証を行なっている．

Tesson らの手法において重要な点は，Coq スクリプトを，プログラム運算でよく見られる次のような記法で記述できることである．

$$\begin{aligned} & f_1 \\ =\ & f_2 \quad \{f_1 = f_2 \text{ が成り立つ理由 }\} \\ =\ & f_3 \quad \{f_2 = f_3 \text{ が成り立つ理由 }\} \\ =\ & \cdots \end{aligned}$$

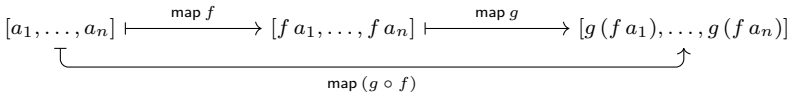
ただし，$f_1, f_2, f_3, \ldots$ は関数プログラムである．こうした記法で Coq スクリプトを記述できることによる恩恵は，単にスクリプトの証明としての可読性が向上することだけではない．こうした記法がサポートされることによって，我々は，「論文に書いてある（あるいは，紙とペンで行なった手書きの）証明をほぼそのまま書き写すだけで，Coq による形式的証明が得

---

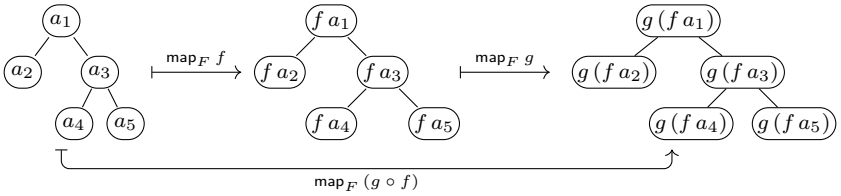
Concurrent Operations on Splay Trees.

Kosuke Murata, 九州工業大学, Kyushu Institute of Technology.

Kento Emoto, 九州工業大学, Kyushu Institute of Technology.




(a) For lists




(b) For binary trees

**図 1**　マップ融合則 $\mathsf{map}\,g \circ \mathsf{map}\,f = \mathsf{map}\,(g \circ f)$

られる」可能性を見出すことができる．一般には，非形式的な証明を形式的証明に移植することは自明な作業ではないが，この手法によって，運算規則の証明の移植に関しては自明になることが期待される．

ところで，Tesson らが検証の対象とした Theory of Lists [2] は，主にリスト関数についての運算規則を集めた教科書である．しかし，実用的な関数プログラミングにおいては，リスト以外にも様々な代数データ型（ADT, Algebraic Data Types）を用いてプログラミングをするであろう．したがって，実用的なプログラム運算の場面を考えると，様々な ADT について成り立つ運算規則である「ADT 汎用な運算規則」が重要であると考えられる．

こうした，ADT に対して汎用的なプログラムの研究は，汎用プログラミング（generic programming）の文脈で盛んに研究されてきた [1]．特に代表的なものは，catamorphism や anamorphism などの再帰図式（recursion scheme）を用いたものであり，これらの再帰図式には様々な運算規則が適用できることも知られている [6]．

再帰図式は，再帰計算の典型的なパタンを捉えたものである．例えば catamorphism は帰納的なデータ構造をたたみ込んで一つの値を得るような再帰的計算，anamorphism はある値から余帰納的なデータ構造を構築するような余再帰的計算を捉えている．

ADT 汎用な再帰図式は，簡単な圏論の概念を用いて述べられる．関手 $F$ に対して，$F$ 始代数や $F$ 終余代数が，様々な代数データ型（あるいは余代数データ型）に対応することはよく知られているが（この解説としては Vene の博士論文 [14] がわかりやすい），再帰図式も同様に，関手 $F$ の選び方によってさまざな代数データ型に対する計算になる．

先に紹介したマップ融合則も，catamorphism を用いて ADT 汎用な規則へと拡張することが可能である．まず，catamorphism を用いると，$f\colon A \to A'$ n い対して，ADT 汎用なマップ関数 $\mathsf{map}_B$ が次のように定義できることが知られている †1:

$$\mathsf{map}_B\ f = (\!|\, \mathsf{in}_{B_{A'}} \circ B(f, \mathrm{id}) \,|\!).$$

ただし，$B$ は双関手で，$B_A = B(A, -)$ とするとき，任意の対象 $A$ に対して $\mu B_A$ が存在するようなものである，この $B$ の選び方によってどの代数データ型に対して動くマップかを決めることができる．こうして定義された ADT 汎用なマップ関数 $\mathsf{map}_B$ でも，次のようなマップ融合則が成り立つ:

$$\mathsf{map}_F\ g \circ \mathsf{map}_F\ f = \mathsf{map}_F\ (g \circ f).$$

直観的なマップ融合則の理解のため，マップ融合則を二分木に適用した例を，図 1 (b) に示す †2．

われわれの研究の目的は，こうした ADT 汎用な運算規則の正しさを証明し，正しく実際のプログラムに適用するためのシステムを構築することである．

†1　この $\mathsf{map}_T$ は実は関手になっているので，データ関手（data functor）と呼ぶこともある．

†2　なお，図 1 (b) においては，$F\ A\ X = \mathbf{1} + A \times X \times X$ である．

本論文の貢献

本論文では，ADT 汎用な運算規則を証明し，Coq プログラムに適用するための手法を提案する．まず，ADT 汎用な運算規則を証明するアイデアは，

- 型クラスを用いて始代数・終余代数などの概念を表現し，
- 型クラスのインスタンスについての量化を用いて，ADT 汎用な定理を表現する

ことである．このアプローチ自体には何の目新しさもないが，

- Tesson らの手法と組み合わせ，さらに少し記法を整理することで，ADT 汎用な運算定理についても，極めてプログラム運算の論文に近いスクリプトが得ることができた

という点が重要であり，この点を実現するために型クラスは重要な役割を果たしている．我々は，

- Histomorphism や futumorphism といった高度な再帰図式を提案した Uustalu らの論文[12] に現れる運算定理をすべて形式化し，可読性の高いスクリプトを得られることを確認した．

形式化の対象として論文[12] を選んだのは，

1. この論文で提案されている histomorphism や futumorphism は，後述するように圏 **Set** で定義できるため Coq で扱うのは易しいが，
2. catamorphism や anamorphism といった単純な再帰図式に比べれば十分複雑で非自明な要素を含んでおり，
3. さらに histomorphism には，動的計画法を用いたフィボナッチ数の計算や，ナップザック問題を解くプログラムなど，重要な実例が知られている

という 3 つの理由からである．

始代数・終余代数の型クラスを用いたもう一つの恩恵は，そのインスタンスを定義することによって，実際に動くプログラムを抽出できることにある．我々は，

- Uustalu らが histomorphism のモチベーティブな例として挙げている動的計画法を用いたフィボナッチ数を計算するプログラムを，histomorphism を用いて Coq 上で定義し，Coq インタプリタ上で動作することを確かめた．

しかし，始代数・終余代数の型クラスのインスタンス化には労力を要する．本論文では，

- 始代数を表す型クラスのインスタンス化を一部自動化するタクティックを提案し，
- 今後の課題として，終余代数を表す型クラスのインスタンス化を自動化するタクティックの構築についてその方針を述べる．

本論文の構成

本論文のこの後の構成は，次のようになっている．第 2 節では，Coq および Tesson らのタクティックライブラリ [10] について述べる．第 3 節では，本論文が形式化する再帰図式について述べる．特に新しい内容はないが，再帰図式について馴染みのない読者の便を考慮し，かなり詳細に述べた．第 4 節では，本論文の理解に必要な範囲で，余帰納的定義について簡単に述べる．第 5 節以降が我々オリジナルの仕事である．第 5 節では，Coq による再帰図式の形式化について述べる．我々の形式化は型クラスを用いている．第 6 節では，型クラスのインスタンス化の例について述べ，第 7 節でその自動化について議論する．第 8 節では関連研究を紹介する．最後に，第 9 節で今度の課題を述べる．

## 2 準備: Coq について

### 2.1 定理証明支援系 Coq

Coq [11] は，対話的定理証明支援系の一つであり，数学や情報科学の定理を機械的に（計算機が理解できる形で）証明するのを補助するソフトウェアである．Coq の中核は Gallina と呼ばれる関数型言語であり，Gallina は，依存型や帰納的・余帰納的な型定義を含む強力な型システムを持っている．Curry-Howard 対応によって，Gallina 言語は型やプログラムを記述するために用いられるだけでなく，命題や証明を記述するのにも用いることができる．

### 2.2 Deep vs shallow embedding

定理証明支援系を用いてプログラムの性質を証明する研究は幅広く行われているが，それらは，deep embedding を採用するものと，shallow embedding を採用するものの二つに大別できる．

前者の deep embedding とは，定理証明支援系の中

核言語をホスト言語とし，そのホスト言語でゲスト言語の意味論を定義した上で，そのゲスト言語の意味論についての性質を，ホスト言語を用いて証明するものである．また後者の shallow embedding とは，定理証明支援系の中核言語で定義したプログラムについての性質を，同一の中核言語を用いて直接証明するものである．Deep embedding は，言語の意味論を自由に定義できるため，様々なプログラミング言語について正確な仕様を記述し証明することが可能になるが，一般には証明にはコストがかかり，また意味論の定義の仕方によっては，プログラム抽出が難しい（ホスト言語上に定義されたゲスト言語のコンパイラを実装した上で，場合によってはさらにそのコンパイラを検証することが必要になってくる）．

以上を踏まえると，deep embedding による検証は，言語設計段階における理論的な検証には適しているが，実際に動作させるプログラムを得ることも視野に入れた実用的な検証には難がある．一方 shallow embedding は，柔軟性には欠けるが，証明のコストが低く，何よりさらに中核言語のインタプリタを利用することで，検証の対象となるプログラムを動作させることが可能である．

本論文は，ユーザが shallow embedding を用いた形式化を与えるための枠組みを提供するものであると言える．

### 2.3 プログラム運算のためのタクティックライブラリ

プログラム運算を扱う論文では，等式変形の過程を記すための，等号を連鎖させた記法が見られる．Tesson ら [10] は，こうした記法を Coq スクリプト上で実現するためのタクティックライブラリを構築した．本論文では，Tesson らの手法を元に我々が論文 [15] で再実装したものを用いた．

## 3 準備: 再帰図式の理論

本節では，再帰図式の理論的解説を行う．本節に新しい内容は特に含まれていないが，再帰図式に不慣れな読者の便宜を図るためと，今後の研究についての議論を充実させるために，かなり詳細な解説を記した．

### 3.1 記法と圏論の基本概念

本論文では，Coq 風の記法を用いる．例えば，型 $A$ 上の恒等関数を表すをラムダ項を $\boldsymbol{\lambda}\,(x : A) \Rightarrow x$ のように書く．

再帰図式は，圏論の用語を使って述べられる．以下では，圏論の基本的な用語を確認する．圏 **Set** は，集合と集合の間の全域写像がなす圏であるとする．

始対象を $\mathbf{0}$，終対象を $\mathbf{1}$ と書く．対象 $X, Y$ の積を $X \times Y$，余積 (和) を $X + Y$ と書く．圏 **Set** においては，$\mathbf{0} = \emptyset$，$\mathbf{1} \cong \{()\}$ であり，

$$X \times Y \cong \{(x, y) \mid x \in X, y \in Y\},$$
$$X + Y \cong \{\mathsf{inl}\ x \mid x \in X\} \cup \{\mathsf{inr}\ y \mid y \in Y\}$$

である．ここで，$\mathsf{inl} \colon X \to X + Y$ および $\mathsf{inr} \colon Y \to X + Y$ は，$X + Y$ 型の値を作るためのコンストラクタである．

二つの関数 $f \colon X \to A$ および $g \colon X \to B$ に対して，関数 $\langle f, g \rangle \colon X \to A \times B$ を，

$$\langle f, g \rangle\ x = (f\ x, g\ x).$$

と定める．また，二つの関数 $f \colon A \to X$ および $g \colon B \to X$ に対して，関数 $[f, g] \colon A + B \to X$ を，

$$\begin{aligned}[f, g]\ x = \mathbf{match}\ x\ \mathbf{with}\\ \mid\ \mathsf{inl}\ a \ \Rightarrow\ f\ a\\ \mid\ \mathsf{inr}\ b \ \Rightarrow\ g\ b\\ \mathbf{end}\end{aligned}$$

と定める．

さらに，二つの関数 $f \colon A \to B$, $g \colon C \to D$ に対して，関数 $f \times g \colon A \times C \to B \times D$ および $f + g \colon A + C \to B + D$ を，

$$(f \times g)\ (a, b) = (f\ a, g\ b)$$
$$(f + g)\ (\mathsf{inl}\ a) = \mathsf{inl}\ (f\ a)$$
$$(f + g)\ (\mathsf{inr}\ b) = \mathsf{inr}\ (g\ b)$$

と定める．

本論文および本論文の形式化では，次の関数外延性を認める:

$$\forall f, g : A \to B, (f = g \iff \forall x \in A, f\ x = g\ x).$$

### 3.2 始代数と終余代数

$\boldsymbol{C}$ を圏とし，$F \colon \boldsymbol{C} \to \boldsymbol{C}$ を関手とする．このとき，$A \in \mathrm{Obj}_{\boldsymbol{C}}$ および射 $\varphi \colon F\ A \to A$ の組 $(A, \varphi)$ を $F$ 代数という．双対として，$A \in \mathrm{Obj}_{\boldsymbol{C}}$ および射 $\varphi \colon A \to F\ A$ の組 $(A, \varphi)$ を $F$ 余代数という．$(A, \varphi)$

が $F$ 代数のとき，単に $\varphi$ を $F$ 代数ということもあり，$(A,\varphi)$ が $F$ 余代数のとき，単に $\varphi$ を $F$ 余代数ということもある．

二つの $F$ 代数 $(A,\varphi)$ および $(B,\psi)$ の間の準同型とは，$\boldsymbol{C}$ の射 $f\colon A \to B$ であって，図式

$$\begin{array}{ccc} F\,A & \xrightarrow{\varphi} & A \\ {\scriptstyle F\,f}\downarrow & & \downarrow{\scriptstyle f} \\ F\,B & \xrightarrow[\psi]{} & B \end{array}$$

を可換にするものである．$F$ 余代数についても，二つの $F$ 余代数の間の準同型が同様に定義される．

$F$ 代数 $(\mu F, \mathsf{in}_F)$ が $F$ 始代数であるとは，任意の $F$ 代数 $(X,\varphi)$ に対して，準同型 $f\colon (\mu F,\mathsf{in}_F) \to (X,\varphi)$ が唯一存在することである．双対的に，$F$ 余代数 $(\nu F,\mathsf{out}_F)$ が $F$ 終余代数であるとは，任意の $F$ 余代数 $(X,\varphi)$ に対して，準同型 $f\colon (X,\varphi) \to (\nu F,\mathsf{out}_F)$ が唯一存在することである．$F\colon \boldsymbol{C} \to \boldsymbol{C}$ に対して，$F$ 始代数および $F$ 終余代数は存在すれば同型を除いて唯一であるから，$F$ 始代数を $(\mu F,\mathsf{in}_F)$，$F$ 終余代数を $(\nu F,\mathsf{out}_F)$ と書く．なお，$F\colon \boldsymbol{C} \to \boldsymbol{C}$ に対して，$\mu F$ は $F$ の最小不動点，$\nu F$ は最大不動点になっている．

Adàmek の定理として，圏 $\boldsymbol{C}$ が始対象を持ち，オメガ鎖

$$A_0 \xrightarrow{f_0} A_1 \xrightarrow{f_1} \cdots$$

が必ず余極限（オメガ余極限）を持つ圏であるとすると，$F\colon \boldsymbol{C} \to \boldsymbol{C}$ がオメガ余極限を保存するならば，$F$ は始代数をもつことが知られている．実際，$\mu F$ は，オメガ鎖

$$\mathbf{0} \xrightarrow{\mathsf{i}_{F(\mathbf{0})}} F(\mathbf{0}) \xrightarrow{F(\mathsf{i}_{F(\mathbf{0})})} F^2(\mathbf{0}) \xrightarrow{F^2(\mathsf{i}_{F(\mathbf{0})})} \cdots$$

の余極限として得られる．ただし，$\mathsf{i}_X$ は $\mathbf{0} \to X$ なる唯一の射のことである．双対的に，圏 $\boldsymbol{C}$ が終対象を持ち，余オメガ鎖が必ず極限を持つ圏であるとき，$F\colon \boldsymbol{C} \to \boldsymbol{C}$ がオメガ極限を保存するとき，$F$ は終余代数をもつ．これにより，$F\colon \mathbf{Set} \to \mathbf{Set}$ が多項式関手であるときには，必ずその $F$ 始代数および $F$ 終余代数をもつことが保証されている．

### 3.3 データ型の表現

Coq の型システムは強正規性を持つので，Coq で定義される関数はどれも全域関数である．したがって，Coq の関数をモデリングするためには，圏 **Set** を考えれば良い．以下，本稿では特に断りがない限り圏 **Set** 上で議論する．

例 **1**（自然数と余自然数） 関手 $N\colon \mathbf{Set} \to \mathbf{Set}$ を，

$$\begin{aligned} N\,X &= \mathbf{1} + X \\ N\,f &= \mathrm{id}_{\mathbf{1}} + f \end{aligned}$$

で定める．$N$ 始代数 $(\mu N, \mathsf{in}_N)$ は，

$$(\mathbf{nat}, [(\boldsymbol{\lambda}\_ \Rightarrow \mathsf{O}), \mathsf{S}])$$

に対応する．ただし，**nat** は自然数の集合

$$\mathbf{nat} = \{\mathsf{O}, \mathsf{S}\ \mathsf{O}, \mathsf{S}\ (\mathsf{S}\ \mathsf{O}), \ldots\}$$

である．また，$N$ 終余代数は，$(\mathbf{conat}, \mathit{pred})$ に対応する．ただし，**conat** は **nat** に，$\mathsf{S}$ が無限に続く元

$$\mathbf{conat} = \mathbf{nat} \cup \{\mathsf{S}\ (\mathsf{S}\ (\mathsf{S}\ \cdots))\}$$

を加えたものであり，また $\mathit{pred}\colon \mathbf{conat} \to \mathbf{1} + \mathbf{conat}$ は，

$$\begin{aligned} \mathit{pred}\ O &= \mathsf{inl}\ () \\ \mathit{pred}\ (\mathsf{S}\ n) &= \mathsf{inr}\ n \end{aligned}$$

である．

例 **2**（リスト） 集合 $A$ に対して，関手 $L_A\colon \mathbf{Set} \to \mathbf{Set}$ を，

$$\begin{aligned} L_A\,X &= \mathbf{1} + A \times X \\ L_A\,f &= \mathrm{id}_{\mathbf{1}} + \mathrm{id}_A \times f \end{aligned}$$

で定義する．$L_A$ 始代数 $(\mu L_A, \mathsf{in}_{L_A})$ は

$$(\mathbf{list}\ A, [\boldsymbol{\lambda}\_ \Rightarrow [\,], (::)])$$

に対応する．ただし，**list** $A$ は $A$ 上のリストを表す型，$[\,]$ は空リスト（$\mathsf{nil}\colon \mathbf{list}\ A$），$(::)\colon A \to \mathbf{list}\ A \to \mathbf{list}\ A$ はリストの先頭に要素を加えるコンストラクタ（$\mathsf{cons}$）である．

例 **3**（ストリーム） 集合 $A$ に対して，関手 $S_A\colon \mathbf{Set} \to \mathbf{Set}$ を，

$$\begin{aligned} S\,X &= A \times X \\ S\,f &= \mathrm{id}_A \times f \end{aligned}$$

で定義する．$S_A$ 始代数は $(\mathbf{0}, \mathrm{id}_{\mathbf{0}})$ に対応し特に面白くないが，一方で $S_A$ 終余代数 $(\nu S_A, \mathsf{out}_{S_A})$ は

$$(\mathbf{Stream}\ A, \langle \mathsf{hd}, \mathsf{tl} \rangle)$$

に対応する．ただし，**Stream** $A$ は $A$ 上のストリーム（無限リスト）の型であり，$\mathsf{hd}\colon \mathbf{Stream}\ A \to A$ は

ストリームの先頭を取り出す関数，$\mathsf{tl}\colon \mathbf{Stream}\ A \to \mathbf{Stream}\ A$ はストリームの尾（先頭を取り除いた残りのストリーム）を取り出す関数である．

### 3.4 再帰図式

再帰図式（recursion sheme）は，再帰的な計算の典型的なパタンを捉えた高階関数である．

#### 3.4.1 Catamorphism

Catamorphism は，帰納的なデータ構造を畳み込んで，一つの値を計算するような計算を捉えたものである．Haskell 言語では，リストの畳み込みを計算する `foldr` とよばれる関数があるが，catamorphism は，`foldr` をリスト以外の様々な代数データ型へ一般化したものであると考えることもできる．$\varphi\colon F\ X \to X$ に対して，$\varphi$ の $F$-catamorphism とは，図式

$$\begin{array}{ccc} F\ \mu F & \xrightarrow{\mathsf{in}_F} & \mu F \\ {\scriptstyle F\ (\!|\varphi|\!)_F}\downarrow & & \downarrow{\scriptstyle (\!|\varphi|\!)_F} \\ F\ X & \xrightarrow[\varphi]{} & X \end{array}$$

を可換にする射 $(\!|\varphi|\!)_F\colon \mu F \to X$ のことである．$F$ が明らかな場合には，$(\!|\varphi|\!)_F$ を単に $(\!|\varphi|\!)$ と書く．

Catamorphism の例として，自然数の畳み込み計算を考えてみよう．$c : \mathbf{1} \to X$ および $f : X \to X$ に対して，$[c, f] : N(X) \to X$ の $N$-catamorphism を考えると，次のようになる:

$$\begin{array}{lcl} (\!|[c,f]|\!)\ \mathsf{O} & = & c\ (), \\ (\!|[c,f]|\!)\ (\mathsf{S}\ n) & = & f\ ((\!|[c,f]|\!)\ n). \end{array}$$

また，リストの場合の畳み込み計算（すなわち，Haskell 言語でいう `foldr`）も，次のように考えることができる．$c : \mathbf{1} \to X$ および $f : A \times X \to X$ に対して，$[c, f] : L_A(X) \to X$ の $L_A$-catamorphism は，

$$\begin{array}{lcl} (\!|[c,f]|\!)\ [\,] & = & c\ (), \\ (\!|[c,f]|\!)\ (a :: xs) & = & f\ (a, (\!|[c,f]|\!)\ xs) \end{array}$$

となる．

自然数とリストのいずれの場合においても，定義の第 2 式の計算は，「一つ前の計算」を用いて定義されていることにも注目されたい．例えば自然数の場合は，$(\!|[c,f]|\!)\ (\mathsf{S}\ n)$ は，「一つ前の計算」$(\!|[c,f]|\!)\ n$ を用いて定義されている．このように，catamorphism は，「一つ前の計算」を使ったごく単純な再帰的計算を記述していると考えることもできる．

Lambeck の補題として，以下のことが知られている．$F$ 始代数 $\mathsf{in}_F\colon F\ \mu F \to \mu F$ には逆射 $\mathsf{in}_F^{-1}\colon \mu F \to F\ \mu F$ が存在し，具体的には次のように逆射を計算できる:

$$\mathsf{in}_F^{-1} = (\!|F\ \mathsf{in}_F|\!)_F.$$

#### 3.4.2 Anamorphism

Anamorphism は catamorphism の双対であり，一つの値から，余帰納的なデータ構造を構築するような計算を捉えたものである．$\varphi\colon X \to F\ X$ に対して，$\varphi$ の $F$-anamorphism とは，図式

$$\begin{array}{ccc} \nu F & \xrightarrow{\mathsf{out}_F} & F\ \nu F \\ {\scriptstyle F\ [\![\varphi]\!]_F}\uparrow & & \uparrow{\scriptstyle F [\![\varphi]\!]_F} \\ X & \xrightarrow[\varphi]{} & F\ X \end{array}$$

を可換にする射 $[\![\varphi]\!]_F\colon X \to \nu F$ のことである．$F$ が明らかな場合には，$[\![\varphi]\!]_F$ を単に $[\![\varphi]\!]$ と書く．

Anamorphism の例として，ストリームを生成する anamorphism を挙げておく．$a\colon X \to A$ および $f\colon X \to X$ に対して，$S_A$-catamorphism $[\![a, f]\!]\colon X \to \mathbf{Stream}\ A$ は，

$$a\ x, a\ (f\ x), a\ (f\ (f\ x)), \ldots, a\ (f^n\ x), \ldots$$

の無限リストである．

Lambeck の補題の双対により，$F$ 終余代数 $\mathsf{out}_F\colon \nu F \to F\ \nu F$ には逆射 $\mathsf{out}_F^{-1}\colon F\ \nu F \to \nu F$ が存在し，具体的には次のように得られることが知られている:

$$\mathsf{out}_F^{-1} = [\![F\ \mathsf{out}_F]\!]_F.$$

#### 3.4.3 より高度な再帰図式の必要性

先に述べた catamorphism と anamorphism は，ごく基本的な再帰図式であり，複雑な再帰計算を捉えるには非力である．例えば，次の $n$ 番目のフィボナッチ数を計算する関数（以下，これをフィボナッチ関数という）$\mathit{fib}\colon \mathbf{nat} \to \mathbf{nat}$ を考える:

$$\begin{array}{lcl} \mathit{fib}\ 0 & = & 0, \\ \mathit{fib}\ 1 & = & 1, \\ \mathit{fib}\ (\mathsf{S}\ (\mathsf{S}\ n)) & = & \mathit{fib}\ (\mathsf{S}\ n) + \mathit{fib}\ n. \end{array}$$

この計算を catamorphism で素直に捉えることは難しい．実際，第 3 式は $\mathit{fib}\ (\mathsf{S}\ (\mathsf{S}\ n))$ を計算するために，「一つ前の計算」$\mathit{fib}\ (\mathsf{S}\ n)$ に加え，「二つ前の計

算」*fib* $n$ を呼び出している．

このような「二つ以上前の計算」を呼び出せるようにする方法は，いくつかある．最もよく知られたものは組化（tupring）である，単に *fib* $n$ を計算する代わりに，連続する二つのフィボナッチ数の組 *pair_fib* $n = (\mathit{fib}\ n, \mathit{fib}\ (\mathsf{S}\ n))$ を計算する関数を定義する．関数 *pair_fib* $n$ は，

$$\begin{aligned}\mathit{pair\_fib}\ 0 &= (0,1)\\ \mathit{pair\_fib}\ (\mathsf{S}\ n) &= f\ (\mathit{pair\_fib}\ n)\\ &\quad \textbf{where}\ f\ (n_0, n_1) = (n_1, n_0 + n_1)\end{aligned}$$

のようなごく単純な再帰で書くことができる．これを基に少し考えると，実は *pair_fib* は次のような $N$-catamorphism で書くことができることがわかる:

$$\mathit{pair\_fib} = (\!|\langle [\boldsymbol{\lambda}\_ \Rightarrow 0, \mathsf{snd}], [\boldsymbol{\lambda}\_ \Rightarrow 1, (+)] \rangle|\!)_N.$$

さらに，$\mathit{fib} = \mathsf{fst} \circ \mathit{pair\_fib}$ であるから，こうしてめでたく *fib* は catamorphism（とアドホックな工夫）で定義できたことになる．

フィボナッチ数は「二つ前までの和」であるが，同様に「三つ前の和」で定義されるトリボナッチ数列 $\{T_n\}_{n=0}^{\infty}$ を，次のような漸化式で定義できる:

$$\begin{aligned}&T_0 = T_1 = 0,\\ &T_2 = 1,\\ &T_{n+3} = T_{n+2} + T_{n+1} + T_n.\end{aligned}$$

トリボナッチ数列を計算する関数プログラムも，確かに単純な組化で書くことができる．四つ前，五つ前でも同じであろう．しかし，この「$n$ つ前」を抽象化した

$$\begin{aligned}&G_{n,0} = \cdots = G_{n,n-2} = 0,\\ &G_{n,n-1} = 1,\\ &G_{n,m} = G_{n,m-1} + \cdots + G_{n,m-n+1}\end{aligned}$$

のような $G_{n,m}$ を計算する関数プログラムを catamorphism で書くことはできるだろうか？ これはもはや非自明な問題である．これは「作為的」な例であるが，こうした「いくつか前」を参照するような計算は珍しくなく，例えばナップザック問題を動的計画法を用いて解く計算などは「いくつ前を参照するかは最初にはわからない」ような計算である．

このように，catamorphism は一般の再帰的計算を捉えるには非力であるため，さらに強力な様々な再帰図式が提案されている．Hylomorphism は，$f\colon X \to F\ X$ および $g\colon F\ Y \to Y$ に対して，

$$[\![f, g]\!]_F = (\!|g|\!)_F \circ [\![f]\!]_F$$

のように定義される再帰図式である．Anamorphism の部分 $[\![f]\!]_F$ で中間データ構造を作り，catamorphism の部分 $(\!|g|\!)_F$ で中間データ構造を消費して一個の値を計算するような計算である．Hylomorphism についてはすでに多くの研究がなされており，例も豊富に知られている上に，hylomorphism に関する様々な運算法則が知られている．特に，fold-build 規則を一般化した酸性雨定理 [9] は，「hylomorphism によって書かれた中間データ構造を生成するようなプログラムから，中間データ構造を生成しないような（そしてそのぶん高速な）プログラムを抽出することができる」という魅力的な運算規則を与えている．

しかし，hylomorphism には，$\mu F$ と $\nu F$ が一致するような圏でしか定義することができないという難点がある．実際，$[\![f]\!]_F\colon X \to \nu F$ と $(\!|g|\!)_F\colon \mu F \to Y$ を合成して $(\!|g|\!)_F \circ [\![f]\!]_F$ を作ることができるのは，$\mu F = \nu F$ となるときだけである．圏 **Set** ではこの仮定は満たされず，cppo（基点つき完備半順序集合）とその上の正則関数の圏 $\mathbf{Cppo}_{\perp}$ のような圏を考えなければならない．先に述べたように，Coq で単純に扱うことのできる関数は全域関数であるため，Coq で素朴に hylomorphism を扱うのは難しいと考えられる．

### 3.5 Histomorphism

Histomorphism は，中間データ構造（主に動的計画法のメモ木）を必要とする計算をうまく扱うための再帰図式の一つであり，hylomorphism と違って圏 **Set** でも扱うことができるため，本論文が目指す Coq による形式化との相性が良い．

Histomorphism が生成するデータ構造は，以下に定義する余自由余モナドとよばれるものである．まず，関手 $F\colon \mathbf{Set} \to \mathbf{Set}$ に対して，$F_A^{\times}\colon \mathbf{Set} \to \mathbf{Set}$ を，次のように定義する:

$$\begin{aligned}F_A^{\times}\ X &= A \times F\ X,\\ F_A^{\times}\ f &= \mathrm{id}_A \times F\ f\end{aligned}$$

この $F_A^{\times}$ を用いて，$F$ の余自由余モナド（cofree comonad）$\widetilde{F}\colon \mathbf{Set} \to \mathbf{Set}$ が，以下のように定義

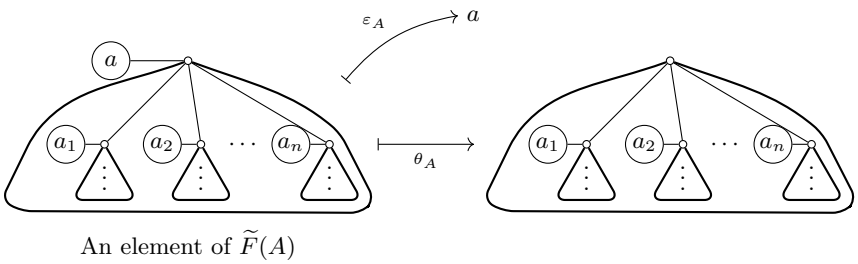



図 **2**　余自由余モナド $\widetilde{F}(A)$ の要素および，$\varepsilon_A$ および $\theta_A$ の概念図

される．

$$\begin{aligned}\widetilde{F}\ A &= \nu F_A^{\times},\\ \widetilde{F}\ f &= [\![\langle f \circ \varepsilon_A, \theta_A\rangle]\!]_{F_A^{\times}},\end{aligned}$$

ただし，

$$\begin{aligned}\varepsilon_A &= \mathsf{fst} \circ \mathsf{out}_{F_A^{\times}} &&: \widetilde{F}\ A \to A,\\ \theta_A &= \mathsf{snd} \circ \mathsf{out}_{F_A^{\times}} &&: \widetilde{F}\ A \to (F \circ \widetilde{F})\ A.\end{aligned}$$

直観的には $\widetilde{F}\ A$ は，$\nu F$ の各ノードに $A$ の要素をアノテーションとして付加したものである．実際，$\widetilde{F}\ \mathbf{1} \cong \nu F$ である．図 2 は，$\widetilde{F}(A)$ および $\varepsilon_A$，$\theta_A$ の概念図であり，図中に現れる $a_1, \ldots, a_n$ はアノテーション値である．$\varepsilon_A$ は $\widetilde{F}\ A$ の各要素の最も上にあるノードに付加されたアノテーション値を返す関数であり，$\theta_A$ は残りの木を返す関数である．

なお，余自由余モナドという名前の通り，$\widetilde{F}\ A$ は余モナドになっているが，その事実は本論文では重要でない．

本題に戻って，histomorphism の定義を述べる．関数 $\varphi\colon (F \circ \widetilde{F})\ A \to A$ に対して，$\varphi$ の histomorphism $\{\!|\varphi|\!\}_F\colon \mu F \to A$ とは，以下を満たす唯一の $\{\!|\varphi|\!\}_F$ である:

$$f \circ \mathsf{in}_F = \varphi \circ F\ [\![\langle f, \mathsf{in}_F^{-1}\rangle]\!]_{F_A^{\times}} \iff f = \{\!|\varphi|\!\}_F. \tag{1}$$

Anamorphism $[\![\langle f, \mathsf{in}_F^{-1}\rangle]\!]_{F_A^{\times}}$ の部分はメモ木を生成するような計算になっており，続く $\varphi$ がメモ木を使って値を計算するような関数になっている．式 1 は，この $\varphi$ から，再帰関数 $\{\!|\varphi|\!\}_F$ を唯一得ることができるという主張している．

問題は，「本当にそんな $\{\!|\varphi|\!\}_F$ が存在するのか」という点であるが，これについては，Uustalu ら [12] によって次のような解が知られている:

$$\{\!|\varphi|\!\}_F = \varepsilon_A \circ (\!|\mathsf{out}_{F_A^{\times}}^{-1} \circ \langle \varphi, \mathsf{id}\rangle |\!)_F.$$

Histomorphism は極めて強い表現力を持っており，例えば動的計画法を用いたフィボナッチ関数 *fib* が図 3 のように定義できる．

### 3.6　その他の再帰図式

ここまでに述べた catamorphism，anamorphism，histomorphism の他にも様々な再帰図式が知られている．例えば，階乗の計算

$$\begin{aligned}\mathit{fact}\ 0 &= 1\\ \mathit{fact}\ (\mathsf{S}\ n) &= n \times \mathit{fact}\ n\end{aligned}$$

のように，$\mathit{fact}\ (\mathsf{S}\ n)$ の定義に引数 $n$ 自身を用いる計算を捉えた recursion scheme として，paramorphism がある．また，apomorphism はその双対である．また，histomorphism の双対として futumorphism が知られている．

## 4　準備: 余帰納的対象の同一性と双模倣

ある多項式関手 $F$ の最大不動点 $\nu F$ として定義されるデータ型は，Coq では余帰納的に定義されるデータ構造に対応している．すなわち，anamorphism の返り値は，通常，余帰納的に定義されたデータ構造である．したがって，Coq における anamorphism は，通常，余帰納的に定義されたデータ構造を返す関数として定義される．

ところで，余帰納的に定義された二つの対象の間の同一性は，Coq での扱いが少し厄介である．というのも，Coq で標準的に定義されている等号は，帰納的に定義されており，余帰納的対象の間の同一性を議

$$
\begin{aligned}
\mathit{fib} \;=\; & \{\!|[\boldsymbol{\lambda}\_ \Rightarrow 1, [(\boldsymbol{\lambda}\_ \Rightarrow 1) \circ \mathsf{snd}, \mathit{add} \circ (\mathrm{id} \times (\mathsf{fst} \circ \mathsf{out}_{F^{\times}_{\mathbf{nat}}}))] \circ \mathit{distl} \circ \mathsf{out}_{F^{\times}_{\mathbf{nat}}}]|\!\}_F \\
& \textbf{where}\ \ \mathit{add}\ (m, n) = m + n, \\
& \qquad\qquad \mathit{distl}\ (a, \mathsf{inl}\ b) = \mathsf{inl}\ (a, b),\ \ \mathit{distl}\ (a, \mathsf{inr}\ c) = \mathsf{inr}\ (a, c).
\end{aligned}
$$

図 3　**Histomorphism** を用いたフィボナッチ関数 *fib* の定義

論するには非力である．その例として，ストリームに対する等式

$$\mathit{ones} = \mathsf{map}\ \mathsf{S}\ \mathit{zeros}$$

を考える．ここで，*zeros* は 0 のみからなる無限ストリーム，*ones* は 1 のみからなる無限ストリームである．一見この等式は自明に成立するように見えるが，Coq ではこの等式を証明することはできない [5]．Coq の等号は，いわゆる Leibniz equality として帰納的に定義されているため，証明に必要な余帰納法を使うことができないからである．

そこで，余帰納的に定義された等号が必要になる．こうした等号は，双模倣を使って次のように定義する方法が知られている [8]．

**定義 4**　$\nu F$ 上の関係 $\sim \subseteq \nu F \times \nu F$ が $F$ 双模倣（$F$-bisimulation）であるとは，以下の図式を可換にする $\gamma\colon \sim \to F(\sim)$ が存在することである:

$$
\begin{array}{ccccc}
\nu F & \xleftarrow{\ \mathsf{fst}\ } & \sim & \xrightarrow{\ \mathsf{snd}\ } & \nu F \\
{\scriptstyle \mathsf{out}_F}\downarrow & & \downarrow{\scriptstyle \gamma} & & \downarrow{\scriptstyle \mathsf{out}_F} \\
F(\nu F) & \xleftarrow[F(\mathsf{fst})]{} & F(\sim) & \xrightarrow[F(\mathsf{snd})]{} & F(\nu F)
\end{array}
\ .
$$

以下の定理は，余帰納法の原理と呼ばれるものである．

**定理 5**　二項関係 $\sim \subseteq \nu F \times \nu F$ が $F$ 双模倣なら，$\sim \subseteq \Delta_{\nu F}$ である．ただし

$$\Delta_{\nu F} = \{(x, x) \mid x \in \nu F\}.$$

**証明**　$(\nu F, \mathsf{out}_F)$ が終余代数であることから，ただちに $\mathsf{fst} = \mathsf{snd}$ が従う．　□

上の命題により，$F$ 双模倣であるような二項関係を，$\nu F$ 上に定義された等号の一つとして使うことが可能である．

以下の二つの命題は，後で Coq において余帰納的に等号を定義する例を示す際に用いる．

**命題 6**　$(S_A, \langle \mathsf{hd}, \mathsf{tl} \rangle)$ を $S_A$ 終余代数とする．$\sim \subseteq \nu S_A \times \nu S_A$ が $S_A$ 双模倣であるための必要十分条件は，任意の $(s, t) \in \sim$ に対して，

$$(\mathsf{hd}\ s = \mathsf{hd}\ t) \wedge (\mathsf{tl}\ s \sim \mathsf{tl}\ t)$$

が成り立つことである．

**証明**　素直に確かめればできる．　□

**命題 7**　$(\widetilde{N}(\mathbf{nat}), \langle \varepsilon, \theta \rangle)$ を $N^{\times}_{\mathbf{nat}}$ 終余代数とする．$\sim \subseteq \widetilde{N}(\mathbf{nat}) \times \widetilde{N}(\mathbf{nat})$ が $N^{\times}_{\mathbf{nat}}$ 双模倣であるための必要十分条件は，任意の $(s, t) \in \sim$ に対して，

$$
\begin{array}{l}
(\varepsilon\ s = \varepsilon\ t) \\
\wedge \left( \begin{array}{l}
\theta\ s = \theta\ t = \mathsf{inl}\ () \\
\vee\ \forall s', t'.\ \left( \begin{array}{l} \theta\ s = \mathsf{inr}\ s' \to \theta\ t = \mathsf{inr}\ t' \\ \to s' \sim t' \end{array} \right)
\end{array} \right)
\end{array}
$$

が成り立つことである．

**証明**　素直に確かめればできる．　□

## 5　Coq による運算規則の形式化

### 5.1　多項式関手の形式化

まず，多項式関手の形式化から始める．図 4 に，多項式関手に関わる概念の Coq による定義を示す．関手は，

- 対象から対象への関数
- 射から射への関数

の二つの関数から成るので，多項式関手を定義するためにはこの二つの関数を定義する必要がある．

まず，多項式関手を帰納的に定義するため，多項式関手の構造を表す AST の型 `PolyF` を定義する．`PolyF` を用いることで，上述の二つの関数を，以下のように帰納的に定義することができる．

- `inst` $F\colon \mathsf{Type} \to \mathsf{Type} \to \mathsf{Type}$ およびその略記 $[\![F]\!]$ は，型（対象）から型（対象）への関数である．
- `fmap` $F\colon (A_0 \to A_1) \to (X_0 \to X_1) \to [\![F]\!]\ A_0\ X_0 \to [\![F]\!]\ A_1\ X_1$ およびその略記 $F(-)[-]$ は関数（射）から関数（射）への関数である．

```
Inductive PolyF : Type :=
    zer : PolyF | one : PolyF | arg1 : PolyF | arg2 : PolyF
 |  Sum   :  PolyF → PolyF → PolyF
 |  Prod  :  PolyF → PolyF → PolyF.

Inductive inst (F : PolyF) (A X : Type) : Type :=
 match F with
     zer ⇒ Empty_set | one ⇒ Unit | arg1 ⇒ A | arg2 ⇒ X
  |  Sum F G   ⇒   (inst F A X) + (inst G A X)
  |  Prod F G  ⇒   (inst F A X) * (inst G A X)
 end.
Notation "⟦F⟧" := (inst F).

Fixpoint fmap (F : PolyF) {A0 A1 X0 X1 : Type}
    (f : A0 → A1) (g : X0 → X1) : ⟦F⟧ A0 X0 → ⟦F⟧ A1 X1
    :=  match F with
          zer ⇒ id | one ⇒ id | arg1 ⇒ f | arg2 ⇒ g
        |   Sum F G ⇒ λx ⇒
              match x with inl x ⇒ inl (fmap F f g x) | inr x ⇒ inr (fmap F g x) end
        |  Prod F G   ⇒   λx ⇒ (fmap f g (fst x), fmap G f g x (snd x))
        end.
Notation "F(g)[f]" := (@fmap F _ _ _ _ g f) (at level 10).
Notation "F[f]" := (@fmap F _ _ _ _ id f) (at level 10).
```

図 4　多項式関手の **Coq** による形式化

```
Lemma fmap_functor_dist :
  ∀  (F : PolyF) {A0 A1 A2 X0 X1 X2 : Type}
     (f0 : A0 → A1) (f1 : A1 → A2) (g0 : X0 → X1) (g1 : X1 → X2),
     F(f1 ∘ f0)[g1 ∘ g0] = F(f1)[g1] ∘ F(f0)[g0].
Lemma fmap_functor_id :
  ∀ (F : PolyF) {A X : Type}, F(@id A)[@id X] = id.
```

図 5　関手が満たすべき性質（の一部）

これらの定義は一見双関手に対応しているが，型 $A$ で添字づけられた関手 $F_A$ を表現していると思うこともできる．実際，$[\![F]\!]\ A$ および $F(@\mathrm{id}\ A)[-]$ のように部分適用すればよい．

例として，$L$ = `Sum one (Prod arg1 arg2)` を考えてみる．型 $A$ に対して，$[\![L]\!]\ A : \mathsf{Type} \to \mathsf{Type}$ は $L_A(X) = \mathbf{1} + A \times X$ を表す．

多項式関手は，図 5 に示すような関手則を満たす．この証明は，$F$ の構造に関する帰納法によって容易にできる．なお，図 5 の `fmap_functor_id` に出現する @id $A$ は，$[\![F]\!]$ の添字 $A$ を型推論するために必要である．

### 5.2　多項式関手および始代数の形式化

次に，始代数および catamorphism の Coq による形式化について述べる．これらの概念は，図 6 に示す型クラス `F_initial_algebra` のように素直に形式化することができる．この型クラスのインスタンス F_initial_algebra $F\ A\ \mu F$ が存在するとき，Coq の関数 $f$: $[\![F]\!]\ A\ X \to X$ に対して，Coq の型と関数の組 $(\mu F, \mathtt{in\_})$ は $[\![F]\!]\ A$ 始代数を表し，Coq の関数 cata $X\ f$ は $f$ の catamorphism を表す．

この定義の下で，catamorphism についての様々な性質を示すことができる．例えば，型コンテキスト

```
Variable  (F : PolyF) (A : Type) (μF : Type)
          (ia : F_initial_algebra F A μF)
```

**Class** `F_initial_algebra` $(F : \mathtt{PolyF})$ $(A : \mathsf{Type})$ $(\mu F : \mathsf{Type})$
$:=$ {

| | | |
|---|---|---|
| `cata` | $:=$ | $\forall\ (X : \mathsf{Type}), (\llbracket F \rrbracket\ A\ X \to X) \to (\mu F \to X);$ |
| `in_` | $:=$ | $\llbracket F \rrbracket\ A\ \mu F \to \mu F;$ |
| `cata_charn` | $:$ | $\forall\ (X : \mathsf{Type})\ (f : \mu F \to X)\ (\varphi : \llbracket F \rrbracket\ A\ X \to X),$ |
| | | $f \circ \mathtt{in\_} = \varphi \circ F[f] \leftrightarrow f = \mathtt{cata}\ X\ \varphi$ |

}.
**Notation** "$(\!|\, f \,|\!)$" := (`cata` _ _ $f$) (at level 5).

図 6　$F$ 代数のための型クラス `F_initial_algebra` の定義

のもとで，以下に示すような命題を証明することができる．

- **Proposition** `cata_cancel` :
  $\forall\ (X : \mathsf{Type})\ (\varphi : \llbracket F \rrbracket\ A\ X \to X),$
  $(\!|\varphi|\!) \circ \mathtt{in\_} = \varphi \circ F[(\!|\varphi|\!)].$
- **Proposition** `cata_refl` : $(\!|\mathtt{in\_}|\!) = \mathrm{id}$
- **Proposition** `cata_fusion` :
  $\forall\ (X\ Y : \mathsf{Type})\ (\varphi : \llbracket F \rrbracket\ A\ X \to X)$
  $(\psi : \llbracket F \rrbracket\ A\ Y \to Y)\ (f : X \to Y),$
  $f \circ \varphi = \psi \circ F[f] \to f \circ (\!|\varphi|\!) = (\!|\psi|\!).$

また，これらの命題を使って，本論文の冒頭で示した次のマップ融合則を示すこともできる:

**Theorem** `map_map_fusion` :
$\forall\ (f : A \to B)\ (g : B \to C),$
$(\mathtt{fmap}\ g) \circ (\mathtt{fmap}\ f) = \mathtt{fmap}\ (g \circ f).$

ユーザは，Coq プログラムに対して，この `map_map_fusion` を適用することにより，ユーザ自身が記述したプログラムに，マップ融合則が「正しく」適用され，プログラムを高速化できるようになることが期待される．

図 7 にマップ融合則を証明する Coq スクリプトを示す．このスクリプトは，この証明が載っている論文 [14] と，ほとんど同様の記法で証明されている．Tesson らの手法 [10] を用いたことにより，等式を連鎖させた記法で書かれていることはもちろんであるが，我々が新たに設計した型クラス `F_initial_algebra` や各種略記によって，$F$-generic なプログラムに対しても元論文のような記法を実現できている．

「証明が論文のような記法で書ける」ことの恩恵は，単にスクリプトの可読性や保守性に止まらない．我々は，ここに「論文に載っている証明をほぼそのまま写せば，Coq による証明が得られるようになる」可能性を見出すことができる．一般に，自然言語で書かれた証明と形式的証明は異なるので，自然言語による証明を Coq が理解できる形に落として形式的証明を得るのは手間である．一方で本論文の形式化においては，**assert** タクティックなどをうまく使うことによって，元論文をほぼそのまま書き写すことで証明を得ることができている．

### 5.3　さらに高度な運算法則の形式化

我々は，catamorphism のみならず，終余代数のための型クラスも定義し，anamorphism の定式化も行った．さらに，histomorphism などの高度な再帰図式についても形式化を行った．例として，histomorphism を表す高階関数 `histo` の定義を，図 8 に示す．この定義において，コンストラクタ `arg1` は，$F_C^{\times}$ の添字 $C$ を表すために用いられている．

表 1 に，我々が Coq で定式化した再帰図式の定義と命題の一覧を示す．これらは，Uustalu らが論文 [12] で提案した定理を全て含んでいる．

## 6　インスタンス化

本節では，二つの型クラス `F_initial_algebra` および `F_terminal_coalgebra` のインスタンス化の例を示す．ここでは，histomorphism を用いて定義されたフィボナッチ関数を，Coq インタプリタ上で動作させるために必要な二つのインスタンスを定義していく．

**Proposition** `map_map_fusion` :
$\forall\ \{A\ B\ C : \mathsf{Type}\}\ (f : A \to B)\ (g : C \to C), (\texttt{fmap}\ g) \circ (\texttt{fmap}\ f) = \texttt{fmap}\ (g \circ f)$.
**Proof**.
**intros**; **unfold** `fmap`.
**assert** $((\texttt{fmap}\ g) \circ \texttt{in\_} \circ F(f)[\mathrm{id}] = \texttt{in\_} \circ F(g \circ f)[\mathrm{id}] \circ F(\mathrm{id})[\texttt{fmap}\ g])$ **as** $H_0$.
{
**Left**
$= ((\!|\ \texttt{in\_} \circ F(g)[\mathrm{id}]\ |\!) \circ \texttt{in\_} \circ F(f)[\mathrm{id}])$.
$= (\texttt{in\_} \circ (F(g)[\mathrm{id}] \circ F(\mathrm{id})[(\!|\ \texttt{in\_} \circ F(g)[\mathrm{id}]\ |\!)]) \circ F(f)[\mathrm{id}])$ {**by** `cata_cancel`}.
$= (\texttt{in\_} \circ (F(g)[(\!|\ \texttt{in\_} \circ F(g)[\mathrm{id}]\ |\!)]) \circ F(f)[\mathrm{id}])$ {**by** `fmap_functor_dist`}.
$= (\texttt{in\_} \circ (F(g \circ f)[(\!|\ \texttt{in\_} \circ F(g)[\mathrm{id}]\ |\!)]))$ {**by** `fmap_functor_dist`}.
$= (\texttt{in\_} \circ F(g \circ f)[\mathrm{id}] \circ F(\mathrm{id})[(\!|\ \texttt{in\_} \circ F(\mathrm{id})[\mathrm{id}]\ |\!)])$ {**by** `fmap_functor_dist`}.
$=$ **Right**.
}
**unfold** `fmap` **in** $H_0$.
**Left**
$= ((\!|\ \texttt{in\_} \circ F(g)[\mathrm{id}]\ |\!) \circ (\!|\ \texttt{in\_} \circ F(f)[\mathrm{id}]\ |\!))$.
$= ((\!|\ \texttt{in\_} \circ F(g \circ)[\mathrm{id}]\ |\!))$ {**apply** `cata_fusion`}.
**Right**.
**Qed**.

図 **7**　マップ融合則を証明する **Coq** スクリプト

**Definition** `histo` $(F : \mathsf{PolyF})\ (\mu F\ C\ \nu FC : \mathsf{Type})\ (ia : \texttt{F\_initial\_algebra}\ F\ C\ \mu F)$
$(tc : \texttt{F\_terminal\_coalgebra}\ (\mathsf{Prod}\ \mathsf{arg1}\ F)\ C\ \nu FC)$
$(\varphi : [\![F]\!]\ C\ \nu FC \to C)$
$:= \mathsf{fst} \circ \mathsf{out\_} \circ (\!|\ \mathsf{out\_inv} \circ \langle \varphi, \mathrm{id} \rangle\ |\!)$.
**Notation** "$\{\!|\varphi|\!\}$" := (`histo` _ _ _ _ _ _ $\varphi$).

図 **8**　**Histomorphism** を表す高階関数 `histo` の定式化

```
Instance Nat_ia (C : Type) : F_initial_algebra (Sum one arg2) C nat :=
  {
    cata X f := fix cataf (n : nat)
    := match n with
       | O    => f (inl ())
       | S n' => f (inr (cataf n'))
       end;
    in_ := [ fun x => O , S ]
  }.
Proof.
  intros X f phi.
  split.
  - intros H; extensionality x; induction x.
    + specialize (equal_f H (inl tt)) as H0; cbv in H0.
      exact H0.
    + specialize (equal_f H (inr x)) as H1; cbv in H1.
      rewrite <- IHx. exact H1.
  - intros H; extensionality x; induction x.
    + rewrite H; induction a; easy.
    + rewrite H; easy.
Defined.
```

図 **9**　**nat** のインスタンス化に伴う証明の例

表 1　**Uustalu, Vene** により提案された再帰図式の **Coq** による定義および定理のステートメント（網掛けされたエントリが定義で，他は定理のステートメント）．誌面の都合により，型コンテキストは省略している．

| | | |
|---|---|---|
| `cata_charn` | : | $f \circ \mathtt{in\_} = \varphi \circ F[f] \leftrightarrow f = (\!\lvert \varphi \rvert\!)$ |
| `cata_cancel` | : | $(\!\lvert \varphi \rvert\!) \circ \mathtt{in\_} = \varphi \circ F[(\!\lvert \varphi \rvert\!)]$ |
| `cata_refl` | : | $\mathrm{id} = (\!\lvert \mathtt{in\_} \rvert\!)$ |
| `cata_fusion` | : | $f \circ \varphi = \psi \circ F[f] \to f \circ (\!\lvert \varphi \rvert\!) = (\!\lvert \psi \rvert\!)$ |
| `lemma1` | : | $\mathtt{in\_} \circ (\!\lvert F[\mathtt{in\_}] \rvert\!) = \mathrm{id} \wedge (\!\lvert F[\mathtt{in\_}] \circ \mathtt{in\_} \rvert\!) = \mathrm{id}$ |
| Def. of $\mathsf{in}^{-1}$ | | $\mathtt{in\_inv} := (\!\lvert F[\mathtt{in\_}] \rvert\!)$ |
| `in_inv_charn` | : | $\mathtt{in} \circ \mathtt{in\_inv} = \mathrm{id} \wedge \mathtt{in\_inv} \circ \mathtt{in\_} = \mathrm{id}$ |
| `ana_charn` | : | $\mathtt{out\_} \circ f = F[f] \circ \varphi \leftrightarrow f = [\!\lvert \varphi \rvert\!]$ |
| `ana_cancel` | : | $\mathtt{out\_} \circ [\!\lvert \varphi \rvert\!] = F[[\!\lvert \varphi \rvert\!]] \circ \varphi$ |
| `ana_refl` | : | $[\!\lvert \mathtt{out\_} \rvert\!] = \mathrm{id}$ |
| `ana_fusion` | : | $\psi \circ f = F[f] \circ \varphi \to [\!\lvert \psi \rvert\!] \circ f = [\!\lvert \varphi \rvert\!]$ |
| Def. of $\mathsf{out}^{-1}$ | | $\mathtt{out\_inv} := [\!\lvert F[\mathtt{out\_}] \rvert\!]$ |
| `out_inv_charn` | : | $\mathtt{out\_inv} \circ \mathtt{out\_} = \mathrm{id} \wedge \mathtt{out\_} \circ \mathtt{out\_inv} = \mathrm{id}$ |
| `lemma2` | : | $f \circ \mathtt{in\_} = \varphi \circ F[\langle f, \mathrm{id} \rangle] \leftrightarrow f = \mathsf{fst} \circ (\!\lvert \langle \varphi, \mathtt{in\_} \circ F[\mathsf{snd}] \rangle \rvert\!)$ |
| Def. of para. | | $\langle\!\lvert \varphi \rvert\!\rangle := \mathsf{fst} \circ (\!\lvert \langle \varphi, \mathtt{in\_} \circ F[\mathsf{snd}] \rangle \rvert\!)$ |
| `para_charn` | : | $f \circ \mathtt{in\_} = \varphi \circ F[\langle f, \mathrm{id} \rangle] \leftrightarrow f = \langle\!\lvert \varphi \rvert\!\rangle$ |
| `para_cancel` | : | $\langle\!\lvert \varphi \rvert\!\rangle \circ \mathtt{in\_} = \varphi \circ F[\langle \langle\!\lvert \varphi \rvert\!\rangle, \mathrm{id} \rangle]$ |
| `para_refl` | : | $\mathrm{id} = \langle\!\lvert \mathtt{in\_} \circ F[\mathsf{fst}] \rvert\!\rangle$ |
| `para_fusion` | : | $f \circ \varphi = \psi \circ F[f \otimes \mathrm{id}] \to f \circ \langle\!\lvert \varphi \rvert\!\rangle = \langle\!\lvert \psi \rvert\!\rangle$ |
| `para_cata` | : | $(\!\lvert \varphi \rvert\!) = \langle\!\lvert \varphi \circ F[\mathsf{fst}] \rvert\!\rangle$ |
| `para_any` | : | $f = \langle\!\lvert f \circ \mathtt{in\_} \circ F[\mathsf{snd}] \rvert\!\rangle$ |
| Def. of apo. | | $[\!\lvert \varphi \rvert\!] := [\!\lvert [\varphi, F[\mathsf{inr}] \circ \mathtt{out\_}] \rvert\!] \circ \mathsf{inl}$ |
| `apo_charn` | : | $\mathtt{out\_} \circ f = F[f, \mathrm{id}] \circ \varphi \leftrightarrow f = [\!\lvert \varphi \rvert\!]$ |
| `apo_cancel` | : | $\mathtt{out\_} \circ [\!\lvert \varphi \rvert\!] = F[[[\!\lvert \varphi \rvert\!], \mathtt{id}]] \circ \varphi$ |
| `apo_refl` | : | $\mathrm{id} = [\!\lvert F[\mathsf{inl}] \circ \mathtt{out\_} \rvert\!]$ |
| `apo_fusion` | : | $\psi \circ f = F[f \oplus \mathtt{id}] \circ \varphi \to [\!\lvert \psi \rvert\!] \circ f = [\!\lvert \varphi \rvert\!]$ |
| `apo_ana` | : | $[\!\lvert \varphi \rvert\!] = [\!\lvert F[\mathsf{inl}] \circ \varphi \rvert\!]$ |
| `apo_any` | : | $f = [\!\lvert F[\mathsf{inr}] \circ \mathtt{out\_} \circ f \rvert\!]$ |
| `lemma3` | : | $f \circ \mathtt{in\_} = \varphi \circ F[(\!\lvert \langle f, \mathtt{in\_inv} \rangle \rvert\!)]$ $\leftrightarrow f = \mathsf{fst} \circ \mathtt{out\_} \circ (\!\lvert \mathtt{out\_inv} \circ \langle \varphi, \mathrm{id} \rangle \rvert\!)$ |
| Def. of histo. | | $\{\!\lvert \varphi \rvert\!\} := \mathsf{fst} \circ \mathtt{out\_} \circ (\!\lvert \mathtt{out\_inv} \circ \langle \varphi, \mathrm{id} \rangle \rvert\!)$ |
| `histo_charn` | : | $f \circ \mathtt{in\_} = \varphi \circ F[[\!\lvert \langle f, \mathtt{in\_inv} \rangle \rvert\!]] \leftrightarrow f = \{\!\lvert \varphi \rvert\!\}$ |
| `histo_cancel` | : | $\{\!\lvert \varphi \rvert\!\} \circ \mathtt{in\_} = \varphi \circ F[[\!\lvert \langle \{\!\lvert \varphi \rvert\!\}, \mathtt{in\_inv} \rangle \rvert\!]]$ |
| `histo_refl` | : | $\mathrm{id} = \{\!\lvert \mathtt{in\_} \circ F[\mathsf{fst} \circ \mathtt{out\_}] \rvert\!\}$ |
| `histo_fusion` | : | $f \circ \varphi = \psi \circ F[[\!\lvert (f \otimes \mathrm{id}) \circ \mathtt{out\_} \rvert\!]] \to f \circ \{\!\lvert \varphi \rvert\!\} = \{\!\lvert \psi \rvert\!\}$ |
| `histo_cata` | : | $(\!\lvert \varphi \rvert\!) = \{\!\lvert \varphi \circ F[(\mathsf{fst} \circ \mathtt{out\_})] \rvert\!\}$ |
| Def. of futu. | | $[\!\lvert \varphi \rvert\!] := [\!\lvert [\varphi, \mathrm{id}] \circ \mathtt{in\_inv} \rvert\!] \circ \mathtt{in\_} \circ \mathsf{inl}$ |
| `futu_charn` | : | $\mathtt{out\_} \circ f = F[(\!\lvert f, \mathtt{out\_inv} \rvert\!)] \circ \varphi \leftrightarrow f = [\!\lvert \varphi \rvert\!]$ |
| `futu_cancel` | : | $\mathtt{out\_} \circ [\!\lvert \varphi \rvert\!] = F[(\!\lvert [[\!\lvert \varphi \rvert\!], \mathtt{out\_inv}] \rvert\!)] \circ \varphi$ |
| `futu_refl` | : | $\mathrm{id} = [\!\lvert F[\mathtt{in\_} \circ \mathsf{inl}] \circ \mathtt{out\_} \rvert\!]$ |
| `futu_fusion` | : | $\psi \circ f = F[(\!\lvert \mathtt{in\_} \circ (f \oplus \mathrm{id}) \rvert\!)] \circ \varphi \to [\!\lvert \psi \rvert\!] \circ f = [\!\lvert \varphi \rvert\!]$ |
| `futu_ana` | : | $[\!\lvert \varphi \rvert\!] = [\!\lvert F[(\mathtt{in\_} \circ \mathsf{inl}) \circ \varphi] \rvert\!]$ |

### 6.1　自然数型のインスタンス化

まず，Coq 標準の自然数型 **nat** が，`F_initial_algebra` のインスタンスであることを示す．具体的には，

```
Instance Nat_ia (C : Type)
  : F_initial_algebra (Sum one arg2) C nat.
```

を定義すれば良い．ここで，`C : Type` は双関手の第一引数であるが，今はまだ使われない．このインスタンスの定義を，図 9 に示す．`nat` の catamorphism や始代数の定義の他に，その定義が catamorphism の特徴付けを満たしていることの証明も求められているが，これはごく単純な帰納法によって完了する．なお，図 9 のスクリプト中に現れる `extensionality` タクティックは，Coq 標準ライブラリの `Coq.Program.Program` の中で定義されている，関数外延性を用いた証明を行

うためのタクティックである．

### 6.2 $N^{\times}_{\mathbf{nat}}$ 終余代数のインスタンス化

次に，*fib* を動かすためのメモ木である $N^{\times}_{\mathbf{nat}}$ 終余代数のインスタンス化を行う．まず，$N^{\times}_{\mathbf{nat}}$ 終余代数は，Coq では次のような余機能的データ型として定義することができる．

**CoInductive** `mid_tree`

:= Nil : **nat** → `mid_tree`

| Cons : **nat** → `mid_tree` → `mid_tree`.

第 4 節で示した命題 7 により，余帰納的に定義された $N^{\times}_{\mathbf{nat}}$ 終余代数のための双模倣関係を，図 10 のように定義することができる．余帰納法の原理から，次の公理を追加する．

**Axiom** `eq_ext` : ∀ ($t_1$ $t_2$ : `mid_tree`),

`EqMidtree` $t_1$ $t_2$ → $t_1 = t_2$.

ここまでの準備のもと，$N^{\times}_{\mathbf{nat}}$ 終余代数 `mid_tree` が `F_terminal_coalgebra` のインスタンスであることを示すことができる．インスタンス宣言を，図 11 に示す．図 11 では残った証明は省略しているが，単純な余帰納法であり，あまり難しくはない．

### 6.3 Histomorphism を動かす

最後に，histomorphism で定義された $n$ 番目のフィボナッチ数を求める関数 *fib* を定義する．図 8 で定義した高階関数 `histo` を用いると，histomorphism によるフィボナッチ関数は図 12 のように定義できる．

図 12 の定義は，以下の論理式を満たすことを，簡単な帰納法によって示すことができる:

**Goal**

`fibo` $0 = 0 \wedge$ `fibo` $1 = 1$

$\wedge \forall n,$ `fibo` (S (S $n$)) = `fibo` (S $n$) + `fibo` $n$.

この性質が成り立っていることにより，histomorphism を用いて定義された `fibo` は素朴なフィボナッチ関数の定義を満たし，したがって意図通りに動作することがわかる．

Coq インタプリタを用いて，この `fibo` を動かすことができる．例えば，**Eval** cbv in `fibo` 6 を実行すると，Coq 処理系は 8 を表示する．また，我々は，この `fibo` の定義から，OCaml や Haskell のプログラムへと抽出できることも確かめた．

## 7 インスタンス化の自動化タクティック

前節では，型クラス `F_initial_algebra` および `F_terminal_coalgebra` のインスタンス化の例を示した．`F_initial_algebra` の場合，

- catamorphism `cata` の定義，
- 始代数 `in_` の定義

の二つの定義を与えた上で，

- `cata_charn` : $\forall f, \varphi,$ $f \circ$ `in_` $= \varphi \circ F\ f \leftrightarrow f =$ `cata` $\varphi$

の証明を書けば良いのであった．その証明の例として，自然数型 `nat` が `F_initial_algebra` のインスタンスであることを宣言する際に必要になる証明を，図 9 に示した．

ところで，よく考えると，3 つめの証明 `cata_charn` は，もちろん `cata`，`in_` として何を与えるかによって難易度が変化するのだが，図 9 の `nat` の例のような場合はほとんど明らかである．そして，`nat` の例のみならず，$\mu F$ が「綺麗に」帰納的に定義されており，`cata` および `in_` が，$\mu F$ の構成に沿って「素直に」定義されている場合は，3 つめの証明は易しい場合も多そうである．実際，← 方向:

$$f \circ \texttt{in\_} = \varphi \circ F\ f \leftarrow f = \texttt{cata}\ \varphi$$

は，つまり

$$\texttt{cata}\ \varphi \circ \texttt{in\_} = \varphi \circ F\ (\texttt{cata}\ \varphi)$$

であり，さらに関数外延性を用いると，任意の $x$: $F\ \mu F$ に対して

$$(\texttt{cata}\ \varphi \circ \texttt{in\_})\ x = (\varphi \circ F\ (\texttt{cata}\ \varphi))\ x$$

を示せば良いことになるが，これは $x$: $F\ \mu F$ に関する帰納法を使って場合分けをした上で，左右の評価を繰り返せば簡単に示すことができそうである．

我々は，この方針に従って，`cata_charn` の ← 方向の証明を自動化するタクティック `auto_instance_onlyif` を，タクティック記述言語 Ltac を用いて図 13 のように実装した．

我々は，このタクティックの有用性を調べるため，図 14 に示す 5 つのテストケースについて，それぞれの `cata_charn` の ← 方向の証明が自動で完遂できる

```
CoInductive EqMidtree (t1 t2 : mid_tree) : Prop :=
| eqmid : ε t1 = ε t2
        → (θ t1 = inl () ∧ θ t2 = inl ())
          ∨(∀ a b, θ t1 = inr a → θ t2 = inr b → EqMidtree a b)
        → EqMidtree t1 t2.
```

図 10　$N^{\times}_{\mathbf{nat}}$ 終余代数のための双模倣関係

```
Definition ε (t : mid_tree) : nat :=
  mathc t with Nil n ⇒ n | Cons n _ ⇒ n end.
Definition θ (t : mid_tree) : nat :=
  mathc t with Nil _ ⇒ inl () | Cons _ t' ⇒ inr t' end.
Instance Mid_tree_tc : F_terminal_coalgebra (Prod arg1 (Sum one arg2))
                                            nat mid_tree
 := { ana X f x := match (f x) with
                   | (n, ux) ⇒ match ux with
                                | inl () ⇒ Nil n
                                | inr x ⇒ Cons n (mid_tree_ana X f x)
                                end
                   end
    out_ := ⟨ε, θ⟩ }.
```

図 11　$N^{\times}_{\mathbf{nat}}$ 終余代数のインスタンス（残った証明は省略している）

```
Definition fibo := {|[λ _ ⇒ 0, [(λ _ ⇒ 1) ∘ snd, λ p ⇒ (fst p + snd p) ∘ (id ⊗ (fst ∘ out_))] ∘ distl ∘ out_]|}
```

図 12　Coq における histomorphism を用いたフィボナッチ関数の定義

```
Ltac auto_instance_only_if :=
  let H := fresh "H" in
  let x := fresh "x" in
  let functor := fresh "functor" in
  (
    intros H; extensionality x; induction x;
    [rewrite H;
     repeat
       ( easy +
         match goal with
         | [ k : inst _ _ _ |- _ ]
           => induction k
         end
       )..]
  ).
```

図 13　タクティック auto_instance_onlyif の実装

か調べた．その結果，どのテストケースにおいても，cata_charn の ← 方向の証明を自動化することが可能であった．

## 8　関連研究

本研究は，主に Tesson らによるタクティックライブラリ [10] を先行研究と位置付けているが，他にも様々な関連研究がある．

まず，プログラム運算の定理証明支援系を用いた形式化という観点からは，例えば Mu ら [7] による Agda を用いた教科書 Algebra of Programming [3] の一部の形式化，Chiang ら [4] による Agda を用いた同教科書の一部の形式化がある．これらは，アレゴリと呼ばれる一種の関係代数を用いたプログラム運算の高度な理論の形式化を与えているが，プログラム抽出までは考慮していない．

## 9　今後の課題

本節では，今後の研究の方向性として，以下の 2 点を挙げておく．

```
Instance Unit_ia (C : Type) : F_initial_algebra (Prod one one) C unit :=
  {
    cata X f := fix cataf (u : unit) := f ((), ());
    in_ := fun _ => ()
  }.

Instance Bool_ia (C : Type) : F_initial_algebra (Sum one one) C bool :=
  {
    cata X f := fix cataf (b : bool)
    := match b with
       | true  => f (inl ())
       | false => f (inr ())
       end;
    in_ := [ fun x => true , fun x => false ]
  }.

Instance Nat_ia (C : Type) : F_initial_algebra (Sum one arg2) C nat :=
  {
    cata X f := fix cataf (n : nat)
    := match n with
       | O    => f (inl ())
       | S n' => f (inr (cataf n'))
       end;
    in_ := [ fun x => O , S ]
  }.

Instance List_ia (A : Type) : F_initial_algebra (Sum one (Prod arg1 arg2)) A (list A) :=
  {
    cata X f :=
      fix cataf (l : list A)
      := match l with
         | nil       => f (inl ())
         | cons a xs => f (inr (a, cataf xs))
         end;
    in_ := [ fun x => nil, fun p => cons (fst p) (snd p) ]
  }.

Instance Tree_ia (A : Type) : F_initial_algebra (Sum one (Prod arg1 (Prod arg2 arg2))) A (Tree A) :=
  {
    cata X f :=
      fix cataf (t : Tree A)
      := match t with
         | Tree_Leaf _          => f (inl ())
         | Tree_Node _ a t1 t2 => f (inr (a, (cataf t1, cataf t2)))
         end;
    in_ := [ fun _ => Tree_Leaf A , fun p => Tree_Node A (fst p) (fst (snd p)) (snd (snd p)) ]
  }.
```

図 14　タクティック auto_instance_onlyif の性能を調べるための 5 つのテストインスタンス

### 9.1　さらに高度な再帰図式の理論の形式化

本論文では，histomorphism などの再帰図式の Coq による形式化を行った．引き続き，さらに高度な再帰図式の理論の形式化を行う．例えば，Uustalu ら [13] は，余モナドから様々な再帰図式が得られることを示しており，さらに汎用的なプログラミングの可能性を示唆している．こうした研究の形式化を行うことは今後の課題である．

### 9.2 cata_charn の → 方向の証明の自動化

第 7 節では，cata_charn の ← 方向の証明を一部自動化するタクティックを提案した．では，cata_charn の → 方向:

$$f \circ \texttt{in\_} = \varphi \circ F\ f \to f = \texttt{cata}\ \varphi$$

の証明を自動化することはできるだろうか．理論的には，仮定に関数外延性を用いて導かれる

$$\forall (t : \mu F),\ (f \circ \texttt{in\_})\, t = (\varphi \circ F\ f \to f)\, t$$

の $t$ に $\texttt{in\_inv}^{-1}\, x$ を入れて計算すれば，結論の

$$\forall (x : F\ \mu F),\ f\, x = \texttt{cata}\, \varphi\, x$$

は得られそうである．しかし，インスタンスの定義時点で in_inv を得ることは難しいため，この方針で自動化タクティックを実装するには，型クラス設計などの修正が必要になると考えられる．これは今後の課題である．

## 参 考 文 献


[1] Backhouse, R. and Gibbons, J.(eds.): *Generic Programming*, Lecture Notes in Computer Science, Vol. 2793, Springer-Verlag Berlin Heidelberg, 2003.

[2] Bird, R.: An introduction to the theory of lists, *Logic of Programming and Calculi of Discrete Design*, Heidelberg, Springer, 1987, pp. 5–42.

[3] Bird, R. and de Moor, O.: *Algebra of Programming*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[4] Chiang, Y.-H. and Mu, S.-C.: Formal derivation of Greedy algorithms from relational specifications: A tutorial, *Journal of Logical and Algebraic Methods in Programming*, Vol. 85(2016), pp. 879–905.

[5] Chlipala, A.: *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*, The MIT Press, 2013.

[6] Meijer, E., Fokkinga, M., and Paterson, R.: Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire, *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, 2000, pp. 124–144.

[7] Mu, S.-C., Ko, H.-S., and Jansson, P.: Algebra of programming in Agda: Dependent types for relational program derivation, *Journal of Functional Programming*, Vol. 19, No. 5(2009), pp. 545–579.

[8] Sangiorgi, D.: *Introduction to Bisimulation and Coinduction*, Cambridge University Press, 2011.

[9] Takano, A. and Meijer, E.: Shortcut Deforestation in Calculational Form, *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, FPCA '95, New York, NY, USA, ACM, 1995, pp. 306–313.

[10] Tesson, J., Hashimoto, H., Hu, Z., Loulergue, F., and Takeichi, M.: Program Calculation in Coq, *Proceedings of the 13th International Conference on Algebraic Methodology and Software Technology (AMAST'10)*, Springer-Verlag, 2011, pp. 163–179.

[11] The Coq development team: *The Coq proof assistant reference manual*, 2018. Version 8.9.0.

[12] Uustalu, T. and Vene, V.: Primitive (Co)Recursion and Course-of-Value (Co)Iteration, Categorically, *Informatica*, Vol. 10(1999), pp. 5–26.

[13] Uustalu, T., Vene, V., and Pardo, A.: Recursion Schemes from Comonads, *Nordic J. of Computing*, Vol. 8, No. 3(2001), pp. 366–390.

[14] Vene, V.: *Categorical Programming with inductive and coinductive types*, PhD Thesis, University of Tartuensis, 2000.

[15] 村田康佑, 江本健斗: 定理証明支援系 Coq における不等式変形記法, 情報処理学会論文誌プログラミング *(PRO) 11(4)*, 2018.