修 士 論 文

定理証明支援系 Coq によるデータ型汎用なプログラム運算

指導教員:　　江本 健斗　准教授

九州工業大学大学院情報工学府
先端情報工学専攻

2019 年度

村田　康佑

論　文　概　要

九州工業大学大学院情報工学府 先端情報工学専攻 知能情報工学専門分野

学生番号	18676022	氏　名	村田　康佑
論文題目	定理証明支援系 Coq によるデータ型汎用なプログラム運算		

1　はじめに

プログラム運算（program calculation）は，プログラムの間に成り立つ運算規則といわれる数学的法則を用いて高速なプログラムを導出する手法であり，関数プログラミングにおける有力な高速化手法である．しかし，ソフトウェアにおいては「効率の良さ」のみならず「正しさの保証」も重要である．それゆえ，運算の途中でバグが入り込まないように，機械的に正しさを検証しながら行うのが望ましい．

Tesson らは，定理証明支援系 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た [1]．しかし，Tesson らの手法では，リスト以外の実用上重要なユーザ定義データ型を扱うプログラムへの適用は難しいという問題がある．

本研究では，Coq 上で**再帰図式**（recursion scheme）を用いたプログラムの運算を支援するライブラリを設計・実装する．再帰図式は，典型的な再帰パタンを捉えたデータ型汎用な高階関数であり，再帰図式を用いることで，任意のユーザ定義データ型に関するプログラムを書くことができる．さらに，再帰図式に関するデータ型汎用な運算規則が数多く知られているため，再帰図式で記述されたプログラムは運算の対象とすることができる．本研究では，表 1 に現れる七つの再帰図式について，これらを用いたプログラムを Coq で記述して動作可能なプログラムを得たり，運算規則を簡単に証明したり，運算規則を安全に適用したりできることを示した．本ライブラリを用いると，運算規則の証明を，図 1 のような等式変形の形で書くことができ，運算規則を示した論文の非形式的証明を，ほぼそのまま書き写すことにより形式的証明を完成させることも可能である．

2　本研究の貢献

本研究の貢献は，具体的には以下の二つである．

第一に，単純な shallow embedding の下で，畳み込み系の三つの再帰図式および反畳み込み系の三つの再帰図式を扱うライブラリを設計・実装した．評価として，提案ライブラリを用いて，論文 [3] に現れる六つの再帰図式およびその運算規則を全て証明することで，論文と極めて似た方針や記法の形式的証明が得られることを確かめた．一方，再畳み込み系の再帰図式である hylomorphism [2] は，この方法では上手く扱うことができないことも明らかにした．

畳み込み	反畳み込み
Catamorphism [2, 3]	Anamorphism [2, 3]
Paramorphism [2, 3]	Apomorphism [3]
Histomorphism [3]	Futumorphism [3]
再畳み込み	
Hylomorphism [2]	

表 1: 様々な再帰図式

```
Proposition map_map_fusion :
  ∀ {A B C : Type} (f : A → B) (g : B → C), (fmap g) ∘ (fmap f) = fmap (g ∘ f).
Proof.
  intros; unfold fmap.
  assert ((fmap g) ∘ in_ ∘ F(f)[id] = in_ ∘ F(g ∘ f)[id] ∘ F(id)[fmap g]) as H0.
  {
    Left
    = ((|in_ ∘ F(g)[id]|) ∘ in_ ∘ F(f)[id]).
    = (in_ ∘ (F(g)[id] ∘ F(id)[(|in_ ∘ F(g)[id]|)]) ∘ F(f)[id])
                                                    {by cata_cancel}.
    = (in_ ∘ (F(g)[(|in_ ∘ F(g)[id]|)] ∘ F(f)[id])   {by fmap_functor_dist}.
    = (in_ ∘ (F(g ∘ f)[(|in_ ∘ F(g)[id]|)]))         {by fmap_functor_dist}.
    = (in_ ∘ F(g ∘ f)[id] ∘ F(id)[(|in_ ∘ F(id)[id]|)]) {by fmap_functor_dist}.
    = Right.
  }
  unfold fmap in H0.
   Left
   = ((|in_ ∘ F(g)[id]|) ∘ (|in_ ∘ F(f)[id]|)).
   = ((|in_ ∘ F(g∘)[id]|))                          {apply cata_fusion}.
   Right.
Qed.
```

図 1: map 融合則の証明（(|–|) は catamorphism を表す）

第二に，hylomorphism を Coq で上手く扱う次策として，Capretta による遅延モナドと Fokkinga および Pardo による効果付き再帰図式を組み合わせた新たな再帰図式である「遅延再帰図式」を定義し，これらが通常の再帰図式と同様の運算規則を満たすことを理論的に確かめた．これにより，Coq で実行可能な hylomorphism の定義を得ることができ，さらに通常の hylomorphism と同じような運算が Coq 上でできることが期待される．本研究では，遅延再帰図式を Coq により実装することで，運算可能なプログラムが得られることを確かめた．

参考文献

[1] Tesson, J., et al.: Program Calculation in Coq, *AMAST 2010, LNCS 6486*, Springer-Verlag, pp. 163–179. (2011)

[2] Meijer, E., et al.: Functional programming with bananas, lenses, envelopes and barbed wire, *FPCA 1991, LNCS 523*, Springer-Verlag, pp. 124–144. (1991)

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

目次

対外発表リスト

◇Refreed Papers (in English)

- Kosuke Murata and Kento Emoto. Recursion schemes in coq. in *Programming Languages and Systems - 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1-4, 2019, Proceedings*. pp. 202–221. (2019) [Mur19]

◇査読付き論文（日本語）

- 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）．Vol. 11, No. 4, pp. 1–12．(2018) [村田 18b]

◇口頭発表・査読なし論文・ポスタ発表

- 村田康佑，江本健斗．高度な運算定理の Coq による証明とその自動化．日本ソフトウェア科学会第 36 回大会，東京．(2019) [村田 19a]
- 村田康佑，江本健斗．Coq を用いた高度なプログラム運算定理の検証に向けて．第 21 回プログラミングおよびプログラミング言語ワークショップ（PPL2019），花巻．(2019) [村田 19b]
- 村田康佑，江本健斗．Coq における検証されたプログラム運算の拡張．日本ソフトウェア科学会第 35 回大会，大阪．(2018) [村田 18a]

第 1 章
はじめに

§1.1 背景

愚直で記述の易しいプログラムは動作が遅くて使い物にならない．一方で，技巧的で高速なプログラムは実装が難しい．例えば，選択ソート（selection sort）は愚直なアルゴリズムであり，少しプログラミングを学習した人であれば誰でも実装できるのであろうが，動作が遅くて大抵の場合は使い物にならない．一方，クイックソート（quick sort）は高速なアルゴリズムであるが，やや技巧的で実装がしにくい．

動作と記述性の両立の難しさこれはプログラムの世界を取り巻く二律背反である．この二律背反に立ち向かい，時に妥協をしながらも，新たな知見を見いだして行くのがプログラミングの研究であると言える．この二律背反に立ち向かう研究は様々あるが，プログラム運算はその一つの成果であると言える．

1.1.1 プログラム運算

プログラム運算（program calculation）[Bir87, Bir97, Mei91, Uus99b, Ven00] は，対象となるプログラムに対してその意味を保存する代数変形を繰り返すことで，より効率の良いプログラムを導出する技術である．愚直でわかりやすいが遅いプログラムから初めて，運算を行うことにより，高速で技巧的なプログラムが得られることが期待される．運算は，全自動での導出を目指すもの（すなわち，ある特定の形で記述されたプログラムに，強力な運算規則を自動適用することで，一気に効率の良いプログラムを得るもの）と，手動での導出でも良しとするもの（すなわち，人間が運算規則を上手く組み合わせて適用することで高速化を行うもの）がある．実用的に重要なのは前者であるが，根本的な思想はどちらも同じであるし，後者の方が理論的な汎用性が高いため，本論文では主に後者の立場で説明する．

プログラム運算の思想を簡単に説明するために，次の問題を考える：「具体的な値 x に対して，

$$f\,x = a_0 + a_1 x + a_2 x^2 + a_3 x^3 \tag{1.1}$$

の値を計算せよ」．ただし，$a_0, \ldots, a_3$ は与えられた定数である．また，$f\,x$ で関数 f を値 x に適用することを表す（一般的な数学では $f(x)$ のように書かれるものである）．さて，この式のまま（特にメモ化などをせずに）素直に値を計算すると，

$$f\,x = a_0 + a_1 \times x + a_2 \times x \times x + a_3 \times x \times x \times x$$

のようになり，3 回の加算と 6 回の乗算を行うことが必要になりそうである．では，次のように

$f\,x$ を等式変形してみよう：

$$\begin{array}{ll} & a_0 + \underline{a_1 x + a_2 x^2 + a_3 x^3} \\ = & \quad \{\ \text{分配則}\ \} \\ & a_0 + x(a_1 + \underline{a_2 x + a_3 x^2}) \\ = & \quad \{\ \text{分配則}\ \} \\ & a_0 + x\big(a_1 + x(a_2 + a_3 x)\big). \end{array} \tag{†}$$

なお，このような式変形の記法は見慣れないものかもしれないが，

$$\begin{array}{ll} & X \\ = & \quad \{\ X = Y\ \text{が成り立つ理由}\ \} \\ & Y \end{array}$$

を鎖状につなげたものである．これはプログラム運算の世界ではよく見られる記法であり，**Dijkstra-Feijen 流**（Dijkstra-Feijen style）[Dij88] とよばれることもある．本論文では，できるだけ Dijkstra-Feijen 流を用いて式変形を提示する．また，本論文独自の記法として，変形に当たって注目した部分には下線を付すこともある[*1]．

さて，本題に戻って，(†) を使って，

$$f\,x = a_0 + x\big(a_1 + x(a_2 + a_3 x)\big) \tag{1.2}$$

と変形してから値を計算すると，素直に計算したとしても，

$$f\,x = a_0 + x \times \big(a_1 + x \times (a_2 + a_3 \times x)\big)$$

となり，3 回の加算と 3 回の乗算で済むことになる．このように，後者の方が乗算を計算する回数が少なくなり，少なくとも計算の効率という点では式 (1.1) よりも式 (1.2) の方が優れていることがわかる．このような式変形は **Honer 則**（Honer's rule）とよばれ，多項式の具体値計算などに古くから用いられている [Hor19].

このように，プログラム運算は，

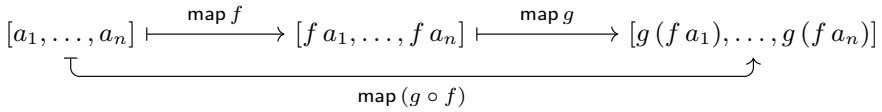
- 愚直でわかりやすいプログラムから始めて（この例で言えば式 (1.1) という単純な多項式から始め），
- 運算規則をうまく適用することによって（この例で言えばうまく x をくくる式変形を繰り返すことによって），
- わかりにくいかもしれないが高速なプログラムを得る（この例で言えば式 (1.2) を得る）

という営みである．

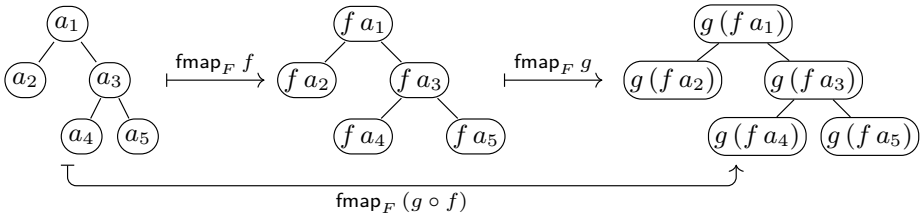
◇実際のプログラム運算　上の例では，多項式をプログラムと見做してプログラム運算を行なったが，実際のプログラム運算は，関数型プログラムの上で運算が行われる．そこでは，関数型プログラムに関する運算規則を用いることになる．そのような運算規則の一つの例は，次の**マップ融合則**（map-fusion law）である：

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

[*1] 本論文では，数式中の下線はすべてこの意味で用いることにする．



(a) リストについてのマップ融合則



(b) リスト以外の木構造についてのマップ融合則（この例では $F\,A\,X = \mathbf{1} + X \times A \times X$）

図 1.1　マップ融合則

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**（higher order function）である：

$$\mathsf{map}\,f\,[a_1, \ldots, a_n] = [f\,a_1, \ldots, f\,a_n].$$

なお，高階関数とは，引数や返り値として関数をとるような関数のことである（分野に依っては，汎関数とよばれることもある）．図 1.1 (a) は，マップ融合則の直観的な説明である．まず，入力となるリスト $[a_1, \ldots, a_n]$ に対して $\mathsf{map}\,f$ を適用することにより，リスト $[f\,a_1, \ldots, f\,a_n]$ が得られ，さらにこれに $\mathsf{map}\,g$ を適用することにより，出力として $[g\,(f\,a_1), \ldots, g\,(f\,a_1)]$ を得ることができる．この出力は，最初の入力であった $[a_1, \ldots, a_n]$ に $\mathsf{map}\,(g \circ f)$ を適用することによっても得られる．

1.1.2　定理証明支援系によるプログラムの正当性検証

プログラムの正当性検証や数学の定理証明といった文脈で，Coq[Coq19] や Isabelle/HOL [Bau01] などの定理証明支援系が使われている．これらのツールは，ユーザーが形式的証明を記述するのを補助し，また型検査器によって証明を検証するものである．典型的な成果としては，四色定理の証明の検証 [Gon08] や C コンパイラの正当性検証 [Ler09] が有名である．また最近，Coq およびその拡張である SSReflect/MathComp についての日本語の教科書 [萩原 18] が出版されるなど，国内でも注目されている．

§1.2　本研究の目的

本研究の究極の目的は，定理証明支援系を用いて，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．

Tesson ら [Tes11] 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[Bir87] に現れる運算規則の検証を行なった．Tesson らの手法において重要な点は，Coq スクリプトを，Dijkstra-Feijen 流の証明で記述できる点である．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．

Tesson らが検証の対象とした Theory of Lists[Bir87] 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，リスト以外にもデータ型を用いることになる．特に，実用的な関数型プログラミングでは，ユーザ定義型を用いてデータ構造を設計することが重要になる [Oka97]．したがって，実用的なプログラム運算の場面を考えると，様々なデータ型について成り立つ運算規則である「データ型汎用な運算規則」が重要である．

データ型汎用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 [Bac03, Gib07a, Gib07b]．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．

再帰図式は，畳み込みや反畳み込みなどの典型的な再帰計算パタンを捉えたデータ型汎用な高階関数群である．図 1.2 は，主要な再帰図式とその定義をまとめたものである．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，始 F 代数や終 F 余代数が，様々なユーザ定義代数データ型（あるいはユーザ定義余代数データ型）に対応することはよく知られているが，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（この詳細は，本論文の第 3 章で述べられる）．

◇データ型汎用なマップ融合則　先に紹介したマップ融合則も，catamorphism を用いてデータ型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，$f\colon A \to A'$ に対して，データ型汎用なマップ関数 fmap_F が次のように定義できることが知られている：

$$\mathsf{fmap}_F\, f = (\!|\, \mathsf{in}_{F_A} \circ F\,(f, \mathrm{id})\,|\!)_F.$$

ただし，F は双関手で，$F_A = F(A, -)$ とするとき，任意の対象 A に対して，始 F_A 代数 $(\mu F_A, \mathsf{in}_{F_A})$ が存在するようなものである．また，$(\!|-|\!)_F$ は catamorphism を表す（詳細な定義は，第 3 章を参照せよ）．この F 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データ型汎用なマップ関数 fmap_F でも，次のようなマップ融合則が成り立つ:

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

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 1.1 (b) に示す．

われわれの研究の目的は，こうしたデータ型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．

畳み込み
（帰納的なデータ型の消費）

Catamorphism[Mal90]
$f = (\!|\varphi|\!)_F \Leftrightarrow f = \varphi \circ F\, f \circ \mathsf{in}_F^{-1}$

Paramorphism[Mee92]
$f = (\!|\varphi|\!)_F \Leftrightarrow f = \varphi \circ F\, \langle f, \mathrm{id}_{\mu F}\rangle \circ \mathsf{in}_F^{-1}$

Histomorphism[Uus99a]
$f = \{\!|\varphi|\!\}_F \Leftrightarrow f = \varphi \circ F\, [\![\langle f, \mathsf{in}_F^{-1}\rangle]\!] \circ \mathsf{in}_F^{-1}$

反畳み込み
（余帰納的なデータ型の生成）

Anamorphism[Mei91]
$f = [\![\psi]\!]_F \Leftrightarrow f = \mathsf{out}_F^{-1} \circ F\, f \circ \psi$

Apomorphism[Vos95]
$f = [\![\psi]\!]_F \Leftrightarrow f = \mathsf{out}_F^{-1} \circ F\, [f, \mathrm{id}_{\nu F}] \circ \psi$

Futumorphism[Uus99a]
$f = [\![\psi]\!]_F \Leftrightarrow f = \mathsf{out}_F^{-1} \circ F\, (\!|[f, \mathsf{out}_F^{-1}]|\!) \circ \psi$

再畳み込み
（中間データ構造を構築した後，それを消費する）

Hylomorphism[Mei91]
$[\![\varphi, \psi]\!]_F = (\!|\varphi|\!)_F \circ [\![\psi]\!]_F$

図 1.2　様々な再帰図式

◇**Deep embedding 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，実行可能なプログラムを得ることが難しい（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である）．

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．それゆえ，本研究ではできるだけ shallow mebedding の下で形式化を行うことにする．

§1.3　本論文の貢献

本研究では，Tesson らの手法を基にして，Coq で再帰図式を扱うためのライブラリを提案する．このライブラリによって，ユーザデータ型を用いたプログラムでも安全に運算を行うことが可能になる．

本論文の具体的な貢献は，以下の 3 点である．

(1) Coq で Dijkstra-Feijen 記法を用いた証明を記述する方法を設計・実装した．この詳細は第 2 章で述べられる．

本手法では，単純な等号のみならず，様々な二項関係を鎖状に繋げた記法を用いることも可能である．特に，等号および自然数上の不等号 $\leq$ を用いた場合について詳細に実装例を示し，評価として Ackermann 関数の諸性質が極めて可読性の高い形式的証明が得られることを確かめた．また，不等号を用いたプログラム運算への応用として，アレゴリを用いた運算についても，運算規則が簡単に記述できることを示した．

(2) 畳み込みおよび反畳み込み系の再帰図式を Coq で扱うための Coq ライブラリを，単純な shallow embedding に基づき設計・実装した．この詳細は第 3 章で述べられる．

この形式化では，再帰図式の中核である始代数や終余代数を表す型クラスを用いることで，データ型汎用性を実現した．提案ライブラリを用いると，ユーザが再帰図式を用いてプログラムを作成し，完成したプログラムに検証された運算規則を適用することで，高速なプログラムを安全に導出することが可能である．また，導出されたプログラムを実行したり，必要に応じて，Coq プログラム抽出機能を用いて他のプログラミング言語で再利用することも可能である．

本ライブラリは貢献(1)で述べた手法を用いており，再帰図式に関わる証明を，プログラム運算で標準的な Dijkstra-Feijen 記法を用いて記述することができる．これによって，プログラム運算の論文に現れる証明をほぼそのまま書き写すことで，形式的証明が得られるという利点がある．例として，我々のライブラリを用いて記述された，データ型汎用なマップ融合則の形式的証明を，図 1.3 に示す．これは，Vene の博士論文 [Ven00] に現れる証明に極めて似た記法の形式的証明になっており，ほぼそのまま書き写すことで形式的証明が得られた例である．

提案ライブラリの評価として，Uustalu らによる論文 [Uus99b] に現れる全ての定義および定理を，論文の証明をほぼそのまま書き写すだけで形式化できることを確かめた．この論文は，histomorphism および futumorphism の提案論文であるが，図 1.2 に挙げた畳み込みおよび反畳み込みの再帰図式の定義および運算規則を幅広く述べている．さらに，再帰図式を用いて運算可能なプログラムが得られることも確かめた．

一方，単純な shallow embedding の限界として，以下の 2 点があげられることも明らかにした：(a) 動作可能なプログラム定義を得るためには，型クラスのインスタンス化を行う必要があり，ユーザに面倒を強いる点，(b) Meijer らによる hylomorphism[Mei91] といった再畳み込みを行う再帰図式の形式化も試みたものの，hylomorphism が本質的に部分関数になりうる事実と，Coq が強正規化性をもつ事実の相性が悪く，うまく形式化できない点．なお，(a) については，自動化タクティックを作成することで，ある程度面倒を緩和できることも明らかにした．

(3) 再畳み込み系の再帰図式を扱う手法として，Capretta による遅延モナド [Cap05] と Pardo による効果付き再帰図式 [Par05] を組み合わせた新たな再帰図式として「遅延再帰図式」を提案し，理論的な整理を行なった．この詳細は第 4 章で述べられる．

遅延再帰図式を用いることで，単純な shallow embedding では扱うことができなかった hylomorphis も Coq で扱うことが可能になることも確認し，実際に遅延再帰図式を用いた Coq

```
Proposition map_map_fusion :
  ∀ {A B C : Type} (f : A → B) (g : C → C), (fmap g) ∘ (fmap f) = fmap (g ∘ f).
Proof.
    intros; unfold fmap.
    assert ((fmap g) ∘ in_ ∘ F(f)[id] = in_ ∘ F(g ∘ f)[id] ∘ F(id)[fmap g]) as H0.
    {
        Left
        = ((| in_ ∘ F(g)[id] |) ∘ in_ ∘ F(f)[id]).
        = (in_ ∘ (F(g)[id] ∘ F(id)[(| in_ ∘ F(g)[id] |)]) ∘ F(f)[id])
                                                          {by cata_cancel}.
        = (in_ ∘ (F(g)[(| in_ ∘ F(g)[id] |)]) ∘ F(f)[id])     {by fmap_functor_dist}.
        = (in_ ∘ (F(g ∘ f)[(| in_ ∘ F(g)[id] |)]))            {by fmap_functor_dist}.
        = (in_ ∘ F(g ∘ f)[id] ∘ F(id)[(| in_ ∘ F(id)[id] |)]) {by fmap_functor_dist}.
        = Right.
    }
    unfold fmap in H0.
     Left
     = ((| in_ ∘ F(g)[id] |) ∘ (| in_ ∘ F(f)[id] |)).
     = ((| in_ ∘ F(g ∘ f)[id] |))                      {apply cata_fusion}.
     Right.
Qed.
```

図 1.3　提案ライブラリを用いた map 融合則の証明

プログラムが動作することを確かめた．この手法では，始代数や終余代数の存在を前提とせず hylomorphism を記述することができ，型クラスのインスタンス化を伴わない汎用プログラミングの可能性も示唆された．一方，遅延再帰図式は通常の再帰図式に比べてやや定義が煩雑なため，得られたプログラムの利用や性質の証明がやや難しくなることも明らかにした．

§1.4　本論文の記法および前提

本論文では，プログラムや関数の定義にあたっては，Haskell 言語または Coq 風の記法を用いるが，可読性の向上のために，ところどころ，数学において標準的な記法も混ぜて用いることにする．また，説明の都合上，章ごとに異なる記法を用いることもあるが，その場合は章ごとに記法を説明しなおす．

本論文では，関数プログラミング（具体的には Haskell 言語や Coq などによるプログラミング）の基本的な知識や，集合や写像といった基本的な数学の知識，領域理論の知識，圏論の知識は説明せずに用いる．以下にこれらを学ぶための基本的な文献をいくつか挙げておく．

関数プログラミング　英語・日本語問わず多くの教科書が容易に入手可能である．定番のものとして，Hutton による教科書（およびその山本による邦訳）[Hut16] を挙げておく．

基本的な数学　英語や日本語問わず多くの文献が容易に入手可能である．日本語の教科書としては，嘉田による教科書 [嘉田 08] を挙げておく．また，Gries らによる教科書 [Gri93] は，ごく基本的な論理や集合を，できるだけ Dijkstra-Feijen 記法を用いて説明した珍しい教科書である．

領域理論　英語では多くの文献が容易に入手可能である．領域理論に特化した教科書としては，Hansen らによる教科書 [Sto94] が定番である．また，Davey らによる教科書 [Dav90] は，順序集

合や束の教科書であるが，領域についても解説があり，丁寧で読みやすい．また，プログラム意味論の教科書である Gunter の教科書 [Gun92] や，プログラム基礎理論の教科書である Mitchell の教科書 [Mit96] はいずれも定番の教科書であり，領域理論の解説も含んでいる．日本語の文献は多くはないが，例えば横内によるプログラム意味論の教科書 [横内 94] には領域理論の丁寧な導入がある．

圏論　英語では多くの文献が容易に入手可能である．圏論の入門的教科書としては，近年は Awodey による教科書 [Awo10] が定番のようである．より計算機科学に特化した内容の圏論の書籍には，例えば Pierce による教科書 [Pie91] がある．また，領域理論の項でも挙げた Gunter の教科書 [Gun92] および Mitchell の教科書 [Mit96]，横内の教科書 [横内 94] にも丁寧な圏論の解説がある．

以上の文献で，本研究で必要になる事柄はほとんどカバーできると思われるが，これらの文献だけではカバーできない道具が必要になる場合は，本文中でその都度参考文献をあげることにする．

§1.5　本論文の構成

これ以降の本論文の構成は，次のようになっている．第 2 章では，Coq での証明を Dijkstra-Feijen 流によく似た記法で書くためのライブラリについて説明する．第 3 章では，畳み込み・反畳み込み再帰図式のための Coq ライブラリについて説明する．第 4 章では，再畳み込み再帰図式のための Coq ライブラリについて説明する．第 5 章では，本論文のまとめや，今後の課題について述べる．

第 2 章
等式・不等式変形のためのタクティックライブラリ

本章では，Dijkstra-Feijen 流に似た鎖状記法を実現するためのタクティックライブラリを設計する手法を提案し，自然数上の等式・不等式を例にして説明する．

また，本手法は，自然数上の等式・不等式のみならず，一般の二項関係に拡張することもできる．そこで，プログラム運算に関わる応用例として，関係代数において部分集合関係 $\subseteq$ を用いて述べられる法則を，鎖状記法を用いて証明する例を示す．

§2.1 はじめに：Coq における自然数上の等式・不等式変形記法

定理証明支援系 Coq での形式的証明は，鉛筆（あるいはボールペン，万年筆，...）でノート上に行う非形式的証明とは異なる記法や構造で記述されるため，数学的な直観がそのまま使えないことも多く，不便である．分量だけを比較しても，形式的証明は，非形式的証明の 4 倍程度である[*1]と言われており，形式的証明と非形式的証明の間には大きな溝があると言える．

本研究が目指す究極の目的は，証明を書く・証明を読むという二つの場面において，非形式と形式の間にあるこの溝を埋めることにある．より具体的な目標は，以下の 2 点について，Coq における形式的証明と非形式的証明の溝を埋めることである．

- **記述性**：鉛筆等で紙上に証明を書くのと同じ感覚で，Coq による形式的証明が作れるようになることを目指す．
- **可読性**：非形式的証明がそうであるように，完成した Coq スクリプトがそれ単体で（Coq と対話することなく）読めるようになることを目指す．

もとより，全ての証明において非形的証明と Coq の形式的証明の差を埋めることは難しい．非形式的証明には，分野ごとに特有の記法や常識があるからである．それゆえ，まずはある特定の分野に絞って，非形式的証明と形式的証明の差異について整理することが必要になってくる．

本研究が対象とするのは，自然数上の等式・不等式である．数学定理やプログラムの性質の形式的証明では，自然数上の等式・不等式についての証明が頻出する．また，自然数や等式・不等式は数学においてごく基本的な対象であるから，主定理が自然数上の等式・不等式ではなかったとしても，その主定理を示すための補題として自然数上の等式・不等式が現れることもある．身近な例としては，計算量の評価といった文脈で等式・不等式上の自然数が重要になることがある [Cor09]．それゆえ，自然数上の等式・不等式について考察することは有用である．

では，自然数上の等式・不等式の証明において，非形式的証明と Coq による形式的証明はどういった違いを孕んでいるだろうか．例題を通して考察する．ここでは，不等式

$$\forall n : \mathbf{nat}\ ,\ 5 \leq n \rightarrow 2^{n+1} \leq n! \tag{2.1}$$

[*1] この 4 という比率は，De Brujin 係数とよばれている．

証明　n についての帰納法による．

(Base case)　$n = 5$ のとき，

$$\begin{aligned} & \text{(左辺)} \\ = \ & 2^{5+1} \\ = \ & 64 \\ \leq \ & 120 \\ = \ & 5! \\ = \ & \text{(右辺)}. \end{aligned}$$

(Induction step)　n の場合を仮定する．すると，$n+1$ の場合も

$$\begin{aligned} & \text{(左辺)} \\ = \ & 2^{(n+1)+1} \\ = \ & 2 \cdot 2^{n+1} \\ \leq \ & 2 \cdot n! && \text{(帰納法の仮定より)} \\ \leq \ & (n+1) \cdot n! && (2 \leq n+1 \text{ より}) \\ = \ & (n+1)! \\ = \ & \text{(右辺)} \end{aligned}$$

より成り立つ．

(a) 非形式的証明

```
1  Lemma sampl_ineq_prop :
2    forall (n : nat),
3      5 <= n -> 2^(n+1) <= n!.
4  Proof.
5    intros n H.
6    induction H.
7    - simpl; omega.
8    - simpl.
9      apply Nat.add_le_mono; auto.
10     rewrite Nat.add_0_r.
11     rewrite <- Nat.mul_1_l at 1.
12     apply Nat.mul_le_mono; auto.
13     omega.
14 Qed.
```

(b) Coq スクリプト

図 2.1　不等式 $\forall n : \mathbf{nat}\ , 5 \leq n \to 2^{n+1} \leq n!$ の典型的な証明

の証明を考える．ただし，$n!$ は n の階乗を表す．この不等式の証明は様々な方法がありうるが，非形式的な証明では図 2.1 (a) のような証明，Coq での形式的証明では図 2.1 (b) のスクリプトによる証明が，それぞれ典型的な例だと考えられる．ただし，図 2.1 (b) のスクリプトに現れる `n!` は，Coq の標準ライブラリでは定義されていない記法であるが，`Notation` コマンドを使って定義したものである．

図 2.1 (a) と図 2.1 (b) を見比べてみると，Coq スクリプトによる形式的証明は，非形式的証明に対して以下のような短所をもっていることに気づく．

- 形式的証明は，非形式的証明に比してより多くの変数が現れており，しかもそれぞれの変数がどういう型を持っているのかがスクリプトからはわからない．このことを定理証明という文脈で言い直せば，証明中に仮定の名前がたくさん現れるが，それぞれの仮定が具体的に何かはわからないということになる．例えば，図 2.1 (b) の形式的証明では `n` の他にも，`H` といった変数（仮定）が現れているが，Coq との対話なしで `H` が何をさしているのか理解するのは難しい．これはスクリプトを証明として読む場合の可読性を大きく下げる原因となっている．
- 非形式的証明では，「左辺を変形して右辺にする」という方針で証明がなされている．そのため，不等式 $L \leq R$ を証明するために，$L = M_1 \leq M_2 = M_3 \leq \cdots \leq M_n = R$ のように項を等号や不等号で鎖状につなげて示す記法が用いられている（以下では，この記法を**鎖状記法**という）．ここでは，$M_1, \ldots, M_n$ といった項に注目をして逐次的変形が進められることによって証明が完成している．一方，形式的証明では，ゴールの不等式全体に注目して，それを

「自明な不等式に還元する」ことで証明を完成させている．例えば，図 2.1 (b) のスクリプトの 8 行目に現れている `apply` タクティックと `auto` タクティックの連続は，`Nat.add_le_mono` ($\forall\, m\ m\ p\ q : \mathbf{nat}\ , n \leq m \to p \leq q \to n + p \leq m + q$) を用いて，現在のサブゴール

$$\Gamma \vdash 2^{m+1} + (2^{m+1} + 0) \leq m! + m \cdot m!$$

を，より簡単な不等式

$$\Gamma \vdash 2^{m+1} + 0 \leq m \cdot m!$$

へと書き換えている．もとより，非形式的証明においてもこうした「自明な不等式に還元する」方法が使われないわけではなく，「左辺を変形して右辺にする」方法と併用されているというべきであろう．しかし，Coq の形式的証明においては鎖状記法はそもそもサポートされていないため，二つの記法が併用できる非形式的証明に比して不自由な思考を強いてしまっていると言える．

以上，Coq における形式的証明の短所を 2 点指摘した．このうち前者については，適切にコメントやアノテーションを施すことによってある程度緩和することができる．例えば `H` という変数が何を指しているかわからなければ，スクリプト上に `H` が何を指しているのかメモしておけば良いということである．

一方，後者はより深刻な問題である．「左辺を変形して右辺にする」という証明の方針をサポートするためには，ぜひとも鎖状記法をサポートするべきである．

◇等式変形についての既存研究　等式変形の記述についての既存研究として，Tesson らのプログラム運算のためのタクティックライブラリ [Tes11] があげられる．Tesson らは，プログラム運算の一つの体系である BMF に基づく等式運算を，Bird によるプログラム運算のレクチャノート [Bir87] に出てくるような形で書くためのタクティックライブラリを与えている．プログラム運算という領域に特化したタクティックライブラリではあるが，一般の等式変形を記述するための枠組みとして使用することもできる．しかし，本論文でサポートするような不等式変形までは具体的にはサポートしていない．

本論文は，Tesson らのアイデアが，等式変形のみならず自然数上の不等式変形にも部分的に適用可能であることを示したものであり，また自然数上の不等式変形で生じる特有の問題について考察を加えたものである．また，本論文で実装したタクティックライブラリは，Tesson らのタクティックライブラリの実装を参考にしつつ，そのアイデアを自然数上の不等式変形へと拡張したものである．

◇本章で述べられる貢献　Tesson らの手法に基づき，不等式の鎖状記法をサポートするためのタクティックライブラリを設計し，タクティック記述言語 Ltac を用いて実装した．このライブラリの使用例として，不等式 (2.1) の証明を記述したスクリプトを図 2.2 に示す．このスクリプトを例にして，本研究で実装したタクティックライブラリの特徴について説明する．このスクリプトの証明では帰納法が用いられているが，base case と induction step のそれぞれについて，左辺から右辺へ至る項の変形の過程が明示的に書かれていることがわかる．特に，図 2.1 (a) の非形式的証明と同様に，

```
1  Lemma sampl_ineq_prop : forall (n : nat), 5 <= n -> 2^(n+1) <= n!.
2  Proof.
3    intros n H.
4    @ H : (5 <= n).
5    induction H.
6    (* base case : n = 5 *)
7    - @ goal : (2 ^ (5 + 1) <= 5!).
8      Left
9      = 64.
10     <= 120   { omega }.
11     = Right.
12   (* induction step *)
13   - @ goal : (2 ^ (S m + 1) <= (S m)!).
14     @ IHle : (2 ^ (m + 1) <= m !).
15     Left
16     = (2 ^ (S m + 1)).
17     = (2 * 2 ^ (m + 1)).
18     <= (2 * m !)          { by IHle }.
19     <= ((S m) * m !)      { because (2 <= (S m)) by omega }.
20     = ((S m)!).
21     = Right.
22 Qed.
```

図 2.2　提案するタクティックライブラリを用いて不等式 $\forall n : \mathbf{nat}\ , 5 \leq n \rightarrow 2^{n+1} \leq n!$ を証明するスクリプト

- ステップごとの変形の結果と，
- ステップごとの変形ができる理由（ただし自明な場合は省略できる）

が明示的に表れている点に注目されたい．例えば，スクリプトの 18 行目から 19 行目は，`IHle` を理由にして，`(2 * (S m + 1)) <= (2 * m !)` なる不等式変形ができることを示している．また，アノテーションによって，現在のサブゴールや重要な変数の型がスクリプト中に明示的に現れていることにも注目されたい．例えば，スクリプトの 8, 14, 15 行目には，現在のサブゴールや変数 `IHle` の型の情報が明示的に書かれている．このアノテーションは単なるコメントではなく，Coq の型チェックを呼び出すタクティックとして設計されており，現在のコンテクストと整合しない記述は型チェックによって弾かれるようになっている．これらの機序によって，Coq を知らないユーザであっても，十分にこのスクリプトは理解できると考えられる．

また本研究では，提案する手法の評価として，実装したタクティックライブラリを用いて，Ackermann 関数の諸性質の形式的証明を，非形式的な証明に似た可読性の高い記法で記述できることを示す．

◇本章の構成　本章のこれ以降の構成は，次のようである．2.2 節では，提案タクティックライブラリの設計と実装について述べる．2.3 節では，ここまで述べた実装では書きにくい証明があることを指摘し，generalized rewriting を用いることで緩和できることを示す．2.4 節では，本ライブラリの評価として，Ackermann 関数に関わる証明を形式化し，手書きの証明に近いスクリプトが得られたことを報告する．2.5 節では，本手法が，プログラム運算では重要な関係代数についての

```
1 Proposition example : forall x y z : nat, x = y -> y <= z -> x <= z.
2 Proof.
3   intros x y z H0 H1.
4   Left
5   = x   { reflexivity }.
6   = y   { exact H0 }.
7   <= z  { exact H1 }.
8   = Right.
9 Qed.
```

図 2.3　核心となるタクティックのみを用いた単純なスクリプトの例

命題の証明へにも応用できることを紹介する．2.6 節では，関連研究を述べる．2.7 節では，まとめと今後の課題について述べる．

§2.2　鎖状記法の設計と実装

本節では，提案する手法の概要について述べる．

2.2.1　タクティックの設計の概要

本研究で提案する手法の核心は，以下の四つの `Tactic Notation`[*2]（以下，単にタクティックという）からなる．

- **タクティック 1**　`Left =` ⟨*term*⟩ `{` ⟨*tactic*⟩ `}`
- **タクティック 2**　`=` ⟨*term*⟩ `{` ⟨*tactic*⟩ `}`
- **タクティック 3**　`<=` ⟨*term*⟩ `{` ⟨*tactic*⟩ `}`
- **タクティック 4**　`= Right`

これらのタクティックを用いて，$\Gamma \vdash t = s$ や $\Gamma \vdash t \leq s$ の形をしたサブゴールを示すことができる．具体的には，**タクティック 1** からはじめて，**タクティック 2**,**3** を繰り返し用い，**タクティック 4** で証明を終える．これらのタクティックを並べると，あたかも項の変形の過程が鎖状に記されているかのように見えるのが重要な点である．図 2.3 は，論理式 $\forall\, x\ y\ z : \mathbf{nat}, x = y \to y \leq z \to x \leq z$ を示すスクリプトであり，このスクリプトの 4 から 8 行目では非形式的証明のような不等式変形が書かれているように見えるが，よくみると上の四つのタクティックを並べているだけであることがわかる．

これらのタクティックは，具体的には次のような動作をする．

- **タクティック 1**　`Left =` ⟨*term*⟩ `{` ⟨*tactic*⟩ `}`

  現在のサブゴール $\Gamma \vdash x \leq y$ を，$\Gamma \vdash \langle term \rangle \leq y$ に書き換える．ただし，その書き換えを行うため，$x = \langle term \rangle$ をタクティック ⟨*tactic*⟩ を用いて示し，それを使って現在のサブゴールに現れる左辺の x を ⟨*term*⟩ へと書き換える．つまり，以下のスクリプトと同じである．

[*2] Coq では，`Tactic Notation` 機能を用いると，タクティック記述言語 Ltac を用いて，新たなタクティックとそのための記法を柔軟に設計することができる．

```
let Hre := fresh "H" in (
  assert (x = ⟨term⟩) as Hre by ⟨tactic⟩; rewrite Hre at 1; clear Hre
).
```

なお，let Hre := fresh "H" in の部分は，一時的におく仮定である $x = \langle term \rangle$ にフレッシュな変数を割り当てるために必要な記述である．

- タクティック 2 = ⟨term⟩ { ⟨tactic⟩ }

タクティック 1 と同様である．

- タクティック 3 <= ⟨term⟩ { ⟨tactic⟩ }

現在のサブゴール $\Gamma \vdash x \leq y$ を $\Gamma \vdash \langle term \rangle \leq y$ に書き換える．ただし，その書き換えを行うため，$x \leq \langle term \rangle$ をタクティック ⟨tactic⟩ を用いて示し，それと $\leq$ の推移律から，サブゴールを $\langle term \rangle \leq y$ に変化させる．つまり，以下のスクリプトと同じである．

```
let Hre := fresh "H" in (
  assert (x <= ⟨term⟩) as Hre by ⟨tactic⟩; transitivity (⟨term⟩) ;
  [ apply Hre | idtac ]; rewrite Hre; clear Hre
).
```

- タクティック 4 = Right

reflexivity の言い換えである．

これらのタクティックは，いずれも Ltac を用いて容易に実装することができる．というのも，Ltac には強力なパタンマッチの機能が用意されており，現在のサブゴールに現れる部分項を取得したり，その部分項を使って既存のタクティックを動かすことができるからである．

再び図 2.3 のスクリプトを例にして，各タクティックの動きについて説明する．このスクリプトを，CoqIDE 上でタクティックごとに読み込んだ様子が図 2.4 である．

- Step 1 は，Coq 標準の intros タクティックを用いているだけであるから問題ないだろう．
- Step 2 では，**タクティック 1** を呼び出している．サブゴールは何も変化していないが，

  assert (x = x) as H2 by reflexivity

  によって H2 : x = x なる項を型コンテキストに追加し，rewrite H2 で書き換えを行なったあと，clear H2 で H2 を削除している．
- Step 3 では，**タクティック 2** を呼び出している．

  assert (x = y) as H2 by (exact H0)

  によって H2 : x = y なる項を型コンテキストに追加し，rewrite H2 で書き換えを行なったあと，clear H2 で H2 を削除している．
- Step 4 では，**タクティック 3** を呼び出している．

  assert (y <= z) as H2 by (exact H1)

  によって H2 : y <= z なる項を型コンテキストに追加し，transitivity z によって，二つのサブゴール “y <= z” と “z <= z” を得ている．前者は exact H0 によって証明を完了し，後者は idtac により何もせずそのままにしておいている．それゆえ，Step4 終了時のゴールとしては，z <= z が残る．

ステップ	スクリプトバッファ	ゴールウィンドウ
0	intros x y z H0 H1. Left = x { reflexivity }. = y { exact H0 }. <= z { exact H1 }. = Right.	============================ forall x y z : nat, x = y -> y <= z -> x <= z
1	intros x y z H0 H1. Left = x { reflexivity }. = y { exact H0 }. <= z { exact H1 }. = Right.	x, y, z : nat H0 : x = y H1 : y <= z ============================ x <= z
2	intros x y z H0 H1. Left = x { reflexivity }. = y { exact H0 }. <= z { exact H1 }. = Right.	x, y, z : nat H0 : x = y H1 : y <= z ============================ x <= z
3	intros x y z H0 H1. Left = x { reflexivity }. = y { exact H0 }. <= z { exact H1 }. = Right.	x, y, z : nat H0 : x = y H1 : y <= z ============================ y <= z
4	intros x y z H0 H1. Left = x { reflexivity }. = y { exact H0 }. <= z { exact H1 }. = Right.	x, y, z : nat H0 : x = y H1 : y <= z ============================ z <= z
5	intros x y z H0 H1. Left = x { reflexivity }. = y { exact H0 }. <= z { exact H1 }. = Right.	

図 2.4　CoqIDE 上で図 2.3 のスクリプトを読み込んだときの対話の様子

- 最後に Step5 で，**タクティック 4** を呼び出している．**タクティック 4** が `reflexivity` を呼び出して，証明を終了している．

なお，こうした素朴な実装では，**タクティック 1** よりも前に**タクティック 2** や**タクティック 3** が呼び出せてしまったり，式変形中に**タクティック 1** が複数回呼び出せてしまうという難点がある．もちろんそうしても正しい形式的証明は得られるが，鎖状記法による証明として人間が読むには意味のわからないものになってしまう．それゆえ，ぜひともこうした (スクリプトを人間が読む上では) 意味のわからないタクティックの適用順を排除する機序が欲しい．実は，これは次節で述べる方法と同じ方法で実現できる．

2.2.2　右辺から左辺に至る記法への対応

前節では，左辺から右辺に至る方向（以下この変形の方向を rightwards という）への変形を記述するタクティックについて説明した．これとほぼ同様の手法で，右辺から左辺に至る方向（以下この変形の方向を leftwards という）への変形を記述するためのタクティックを実装することができる．具体的には，**タクティック 1**，**タクティック 3**，**タクティック 4** の代わりに以下の三つのタクティックを実装し，**タクティック 2** を左辺でなく右辺を書き換えるようにすればよい．

- **タクティック 5** `Right =` ⟨*term*⟩ `{` ⟨*tactic*⟩ `}`

現在のサブゴールが $\Gamma \vdash x \leq y$ のとき，$y = \langle term \rangle$ を $\langle tactic \rangle$ により示し，右辺 y を $\langle term \rangle$ へ書き換える．

- **タクティック 6** `>=` $\langle term \rangle$ `{` $\langle tactic \rangle$ `}`
  現在のサブゴールが $\Gamma \vdash x \leq y$ のとき，$y \leq \langle term \rangle$ を $\langle tactic \rangle$ により示し，$\leq$ の推移律を用いて右辺 y を $\langle term \rangle$ へ書き換える．
- **タクティック 7** `= Left`
  `reflexivity` と同様．

ところが，問題となるのは rightwards な変形と leftwards な変形を同時にサポートしたい場合である．というのも，**タクティック 2** を呼び出した時点での現在の変形の方向が，leftwards か rightwards かどちらだったか覚えていないと，左辺と右辺のどちらを書き換えたら良いかわからなくなるからである．

タクティックを正しい順序で呼び出している限り，rightwards な変形の最初は**タクティック 1** で始まるし，leftwards な変形の最初は**タクティック 5** で始まる．それゆえ，**タクティック 1** および**タクティック 5** が呼び出された時点で，変形の向きが rightwards か leftwards かを表すダミー項を追加しておけば，書き換えの方向を覚えておくことができる．そのダミー項のための型として，二つのデータ型

```
Inductive state : Type := Rightwards : state | Leftwards : state.
```

および

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

を定義しておく．その上で，**タクティック 1** が呼び出されたら，タクティック

```
pose ( memo Rightwards ).
```

を呼び出すようにすれば，型コンテクストに

```
P := memo Rightwards : Prop
```

という仮定が追加される．**タクティック 5** が呼び出された場合も同様に `memo Leftwards` なる項を型コンテキストに追加する．そうしておけば，Ltac のゴールのパタンマッチを用いて，型コンテクストに `memo Rightwards : Prop` があるか `memo Leftwards : Prop` があるか調べることで現在の書き換えの向きがわかるわけである．それゆえ，**タクティック 2** を変形の向きに応じて書き換える向きを変えたり，**タクティック 3** と**タクティック 4** を rightwards な書き換えの時にしか呼び出せないようにするといったことが可能になる．

2.2.3 狭義の不等号

これまで，広義の不等号（すなわち，$\leq$ と $\geq$）における鎖状記法について議論した．しかし，ゴールが $L < R$ のような狭義の不等号（すなわち，$<$ と $>$）による不等式になっている場合について考えると，これまでのタクティックだけでは対応できない．

ひとまず rightwards な変形のみを考える．狭義の不等号による不等式を鎖状記法により証明するために，以下の 2 点を拡張することが必要である．

(1) **タクティック 2** を，$<$ にも対応できるように拡張する．具体的には，**タクティック 2** について，さらに以下のルールを追加すればよい：現在のサブゴールの $\Gamma \vdash x < y$ を $\Gamma \vdash \langle term \rangle < y$ に書き換えるようにする．ただし，その書き換えを行うため，$x < \langle term \rangle$ をタクティック $\langle tactic \rangle$ を用いて示し，それと $<$ の推移律から $\langle term \rangle < y$ を新たなサブゴールにする．

(2) 新たなタクティック

`<` $\langle term \rangle$ `{`$\langle tactic \rangle$`}`

を実装する．このタクティックは，以下のような動作をする：現在のサブゴール $\Gamma \vdash x < y$ を $\Gamma \vdash \langle term \rangle \leq y$ に書き換えるようにする．ただし，その書き換えを行うため，$x < \langle term \rangle$ をタクティック $\langle tactic \rangle$ を用いて示し，それと，

$$x < \langle term \rangle \to \langle term \rangle \leq y \to x < y \tag{2.2}$$

が成り立つことから，$\langle term \rangle \leq y$ を新たなサブゴールにする．

上の 2 点のうち 2 点目について，「現在のサブゴール $\Gamma \vdash x < y$ を，($\Gamma \vdash \langle term \rangle < y$ ではなく) $\Gamma \vdash \langle term \rangle \leq y$ にする」という点が重要である．つまり，このタクティックが呼び出されたおかげで，ゴールに現れる不等号を $<$ から $\leq$ へと変えるわけである．というのも，広義の不等号は `reflexivity` が成り立たないので，$<$ のままでは**タクティック 4** (`= Right.`) で証明を終わることができないからである．幸いにして，(2.2) 式は常に成り立つので，いつでもこのような書き換えを行うことができる．

2.2.4 自明なタクティックの省略

図 2.3 のスクリプトの 5 から 6 行目において，`x = x` を示すためのタクティックとして `reflexivity` タクティックを指定しているが，この記述は冗長である．なぜならば，`x = x` を示すために `reflexivity` を用いるのは当たり前だからである．このように適用するタクティックが自明な場合には，{$tactic$} 部を省略できるようにしたい．

そのためには，**タクティック 1** に加えて，新たに

- **タクティック 1′** `Left =` $\langle term \rangle$

を実装すれば良い．この**タクティック 1′** は，ほとんど**タクティック 1** と同様だが，$x = \langle term \rangle$ の証明を $\langle tactic \rangle$ で行うのではなく，例えば

```
(simpl;reflexivity) + easy + auto
```

で行うようにすればよい．これは，`simpl;reflexivity`，`easy`，`auto` を順番に試すものである．これで自明なタクティックは省略できるようになる．

また，**タクティック 1** の $\langle tactic \rangle$ の部分には，以下のパタンもよく現れる．

- `rewrite` H
- `assert` t `as` H `by` $\langle tactic \rangle$`; rewrite` H`; clear` H

これらについても，**タクティック 1** の実装の $\langle tactic \rangle$ 部分を上記に置き換えた以下のようなタク

ティックを用意しておくと便利である．

- タクティック 1″ `Left =` ⟨*term*⟩`{ by` ⟨*tactic*⟩ `}`
- タクティック 1‴ `Left =` ⟨*term*⟩`{ because` t `by` ⟨*tactic*⟩ `}`

タクティック 1 のみならず，**タクティック 2**，**タクティック 3**，**タクティック 6** についても，同様に ⟨*tactic*⟩ 部を省略したタクティックを作ることができる．

2.2.5 強力な自動証明タクティックとの併用

本研究で提案するライブラリは，強力な自動証明タクティックと併せて使用することによって，変形の 1 ステップを人間が理解しやすい程度に大きなものにできるため，よりわかりやすい証明を記述できるようになることが期待される．具体的には，以下のようなタクティックと併用すると良い．

omega タクティック　`omega` タクティックは，加法のみからなる（したがって乗法を含まない）自然数上の等式や不等式[*3]を自動証明するためのタクティックである．

ring タクティック　`ring` タクティックは，環（ring）や半環（semiring）上の多項式の等式を自動的に証明するタクティックである．Coq での自然数型 `nat` は半環であるため，自然数上の多項式についての等式は `ring` タクティックを用いて自動的に証明できる．

2.2.6 アノテーション

2.1 節で指摘したように，スクリプトの可読性を向上するために，現在のゴールや重要な変数の型などをスクリプト中に明示しておくためのアノテーションが導入されるべきである．そこで，以下の二つのアノテーションを用意した．

- `@` H `:` t
- `@ goal :` t

上は変数 H の型が t であることを表し，下は現在のサブゴールが t であることを表す．このアノテーションも Ltac による `tactic notation` として実装することができる．具体的には，図 2.5 のようにすればよい．この実装では，パタンマッチによってアノテーションの記述が現在のコンテクストにあっているかどうかを検査し，あっていなければ適切なエラーメッセージを表示して失敗するようになっている．

2.2.7 Tesson らの先行研究との差異

Tesson らのライブラリは，標準的な Dijkstra-Feijen 流を採用しており，例えば $f_0 = f_1$ を示す場合には，

$$\begin{array}{ll} & f_0 \\ = & \quad \{\ f_0 = f_1 \text{ が成り立つ理由 }\} \\ & f_1 \end{array}$$

[*3] より厳密には，量化子のない Presburger 算術の論理式として表せる範囲のものということになる．

```
1  Tactic Notation (at level 2) "@" ident(H1) ":" constr(t) :=
2    match goal with
3    | [ H : t |- _ ] =>
4      match H with
5      | H1 => idtac "Type Annotation:" ; idtac "OK, "H":"t"is in current environment."
6      | _  => fail 2 "Type Annotation Error: No such identifier"H1"that is typed"t
7      end
8    | _ => fail 2 "Type Annotation Error: No such identifiers that is typed"t
9    end.
10
11 Tactic Notation (at level 2) "@" "goal" ":" constr(t) :=
12   match goal with
13   | [ |- t ] => idtac "OK, current goal is"t
14   | [ |- ?t1 ] => fail 2 "Current goal is not"t", but is"t1
15   end.
```

図 2.5　アノテーションの実装

のような，$=$ の直後に変形ができる理由を書く流儀になっている．しかし，本研究では，標準的な数学の教科書でもしばしば見られる流儀として，

$$\begin{aligned} & f_0 \\ = \ & f_1 \qquad \{\ f_0 = f_1 \text{ が成り立つ理由 }\} \end{aligned}$$

のように，f_1 の後に変形ができる理由を書く流儀も採用できるようにしている．どちらの流儀の実装も同じように実装できるし，両方の流儀を同時にサポートすることも簡単であるが，本論文では，後者の流儀を使って説明した．これには大きな理由はないが，強いて言えば，後者の方が，Proof General を用いたときの Emacs の自動インデントルールと相性が良いと思われるためである[*4]．

§2.3　Generalized rewriting の利用

本章では，generalized rewriting [Soz10] を用いると，鎖状記法を用いた証明記述がより直観的になる場合があることを説明する．

2.3.1　問題提起

突然だが，図 2.6 のスクリプトは，

$$\forall x\ y : \mathbf{nat}\ ,\ x \leq y \to \mathsf{S}\ (\mathsf{S}\ (\mathsf{S}\ (x^2))) \leq \mathsf{S}\ (\mathsf{S}\ (\mathsf{S}\ (y^2)))$$

の証明である．しかし，12 行目の `(* ??? *)` の部分が欠けていて不完全である．この部分には，$H : x \leq y$ を仮定として

$$\mathsf{S}\ (\mathsf{S}\ (\mathsf{S}\ (x^2))) \leq \mathsf{S}\ (\mathsf{S}\ (\mathsf{S}\ (y^2))) \tag{2.3}$$

[*4] Murata ら [Mur19] は，後者の流儀の方が理由の省略と相性が良いと述べているが，Coq の Tactic Notation では改行が無視されるため，理由を省略したらどちらの流儀もまったく同じ記法になると考えられる．

```
Proposition example :
  forall (x y : nat) ,
    x <= y ->
    S (S (S (x^2))) <= S (S (S (y^2))).
Proof.
  intros x y H.
  @ goal :
    (S (S (S (x^2))) <= S (S (S (y^2)))).
  @ H : (x <= y).
  Left
  =  (S (S (S (x^2)))).
  <= (S (S (S (y^2)))) { (* ??? *) }.
  = Right.
Qed.
```

図 2.6　例題: $\forall x\ y : \mathbf{nat}\ ,\ x \leq y \to \mathsf{S}\ (\mathsf{S}\ (\mathsf{S}\ (x^2))) \leq \mathsf{S}\ (\mathsf{S}\ (\mathsf{S}\ (y^2)))$ を証明するスクリプト (ただし一部欠けていて不完全である)

を示すタクティックを記述することになるが，何を記述するのが良いだろうか？　もちろん，乗算を含む不等式なので omega タクティックは使うことができない．

通常，(2.3) 式のような不等式を示すとき，多くの Coq ユーザーは，

- $\forall x\ y : \mathbf{nat}, x \leq y \to \mathsf{S}\ x \leq \mathsf{S}\ y$
- $\forall x\ y : \mathbf{nat}, x \leq y \to x^2 \leq y^2$

のような命題の apply を繰り返して，仮定 $H : x \leq y$ へと帰結させるのではないだろうか．この方針で証明を書くとすれば，仮に上の二つの命題を S_monotone と sq_monotone として，(* ??? *) の部分に以下のようなタクティックを書くことになる．

```
repeat apply S_monotone; apply sq_monotone; exact H
```

しかし，これはまさに (2.3) 式の証明を構築するスクリプトそのものであり，もとの鎖状記法の中にこのスクリプトを書くのは不自然である．

直観的には，2 乗 $(-)^2$ も後者関数 S も $\leq$ に関して単調であるのだから，そのまま x を y に書き換えたい．実は，そうした直観的な証明を Coq 上で書く方法がある．端的には generalized rewriting を用いれば良いのだが，次節以降でそれを詳しく説明する．

2.3.2　Generalized rewriting

Coq の rewrite タクティックは，Coq のデフォルトの等号 = が Leibniz equality であることを利用して，t = s であるときゴールに現れる t を s に書き換えるものである．実はこの rewrite タクティックは，Coq 標準ライブラリにある Setoid モジュールを読み込むことで，等号のみならず様々な二項関係へ一般化して使うことができる．様々な二項関係といっても，同値関係へと一般化して使用するのが普通であるが，実際には推移的な関係であれば同値関係である必要はないし，本研究でも $\leq$ といった同値関係でない関係に拡張する．

Generalized rewriting のアイデアについて，本研究に関係する範囲で簡単に説明する．そのた

めに，まずいくつか用語の定義をする．A, B を型とする．A 上の 2 項関係 R および $m : A$ に対して，m が R の morphism あるいは proper element であるとは，$R\ m\ m$ が成り立つことである．また，A 上の 2 項関係 R および B 上の 2 項関係 R' に対して，$A \to B$ 上の 2 項関係 $R \mathrel{+\!\!+\!\!\to} R'$ を以下のように定義する：

$$\boldsymbol{\lambda}\ (f\ g : A \to B),\ \forall\ (x\ y : A),\ R\ x\ y \to R'\ (f\ x)\ (g\ y).$$

特に，関数 $f : A \to B$ が，$A \to B$ 上の 2 項関係 $R \mathrel{+\!\!+\!\!\to} R'$ の morphism であるとは，

$$\forall\ x\ y,\ R\ x\ y \to R'\ (f\ x)\ (f\ y)$$

が成り立つことである．例えば，後者 S は $\leq \mathrel{+\!\!+\!\!\to} \leq$ の morphism である．なぜならば，

$$\forall\ (x\ y : \mathbf{nat}),\ x \leq y \to \mathsf{S}\ x \leq \mathsf{S}\ y$$

が成り立つからである．

A を型とし，R を推移的な二項関係とする．ひとたび関数 $f : A \to A$ が $R \mathrel{+\!\!+\!\!\to} R$ の morphism だと示されると，以下の二つの推論ができるようになる：

$$\frac{\Gamma,\ R\ x\ y \vdash R\ (f\ y)\ t}{\Gamma,\ R\ x\ y \vdash R\ (f\ x)\ t} \qquad \frac{\Gamma,\ R\ x\ y \vdash R\ t\ (f\ x)}{\Gamma,\ R\ x\ y \vdash R\ t\ (f\ y)}$$

この推論が妥当であることは容易に示せる．重要な点は，上の二つの推論について，左の推論は $R\ x\ y$ を理由にしてゴールの $R\ (f\ x)\ t$ を $R\ (f\ y)\ t$ に書き換えることができると言っていると読めるし，右の推論は $R\ x\ y$ を理由にしてゴールの $R\ t\ (f\ y)$ を $R\ t\ (f\ x)$ に書き換えることができると言っていると読めるという点である．

上の規則について，わかりやすい例を挙げておく．$\leq$ は推移的な二項関係であるし，後者関数 S は $\leq \mathrel{+\!\!+\!\!\to} \leq$ の morphism であったから，

$$\frac{\Gamma,\ x \leq y \vdash \mathsf{S}\ y \leq t}{\Gamma,\ x \leq y \vdash \mathsf{S}\ x \leq t} \qquad \frac{\Gamma,\ x \leq y \vdash t \leq \mathsf{S}\ x}{\Gamma,\ x \leq y \vdash t \leq \mathsf{S}\ y} \tag{2.4}$$

なる書き換えが可能である．この妥当性は直観的に明らかだろう．

2.3.3 Coq における gneralized rewriting

Coq では，標準ライブラリの `Coq.Classes.Morphisms` 上に，morphism (proper element) と $\mathrel{+\!\!+\!\!\to}$ に対応する概念として，それぞれ `Proper` 型クラスと，`++>` 演算子が定義されている．それゆえ，例えば，後者関数 S が $\leq \mathrel{+\!\!+\!\!\to} \leq$ の morphism であることを，`Proper (le ++> le) S` のインスタンスをつくる形で宣言することができる[*5]．具体的には，以下のスクリプトのようにして宣言できる．

```
Program Instance morphism_S_le :
  Proper (le ++> le) S.
Next Obligation.
Proof.
  simpl_relation.
Qed.
```

[*5] $\leq$ は，Coq 上では関係 `le` として定義されている．

このインスタンスを作成した後では，rewrite タクティックを用いて，(2.4) 式のような書き換えが可能になる．例えば，サブゴールが

```
x, y : nat
H : x <= y
============================
S (S (S x)) <= S (S (S y))
```

という状態で rewrite H を読み込むと，このサブゴールは

```
x, y : nat
H : x <= y
============================
S (S (S y)) <= S (S (S y))
```

へと変化する．すなわち，x が y に書き換わっているのである．なお，この例から分かるように，morphism が複数回適用されていても書き換えることが可能である．

2.3.1 節の最初の質問に戻る．図 2.6 の (* ??? *) に当てはまるタクティックは何が良いかという問題であったが，以下の二つのインスタンスが作成されていれば，“compute; rewrite H” と書くだけで済ませることができる．

- Proper (le ++> le) S
- Proper (le ++> le) (fun x => x^2)

なお，compute タクティックはゴールをできるだけ簡約するタクティックである．上のインスタンスにおける (fun x => x^2) をゴール中の x^2 にパタンマッチさせるために，compute タクティックを用いて簡約する必要がある．compute タクティックを用いなければならないのはユーザにとって冗長であるが，2.2.4 節で述べたのと同様な方法で，タクティックライブラリ側で適切な省略記法を定義して隠蔽すれば，大した問題ではない．

2.3.4 提案タクティックライブラリでの generalized rewriting の利用

本研究で実装したタクティックライブラリには，自然数上の不等式の証明でよく使う morphism について，あらかじめインスタンスを作成している．それゆえ，1 章で提示した図 2.2 のスクリプトにおいて，17 から 18 行目の変形および 19 から 20 行目の変形を簡潔に記述することができている (なお，該当箇所に現れている by··· や because···by··· については，2.2.4 節を参照)．

§2.4 応用例: Ackermann 関数の性質

本章では，情報科学において基本的な命題の証明を通して，本手法の有用性を検証する．具体的には，提案したタクティックライブラリを用いて Ackermann 関数の性質をいくつか示した．結果として，標準的な教科書（例えば，有川による教科書 [有川 86]）に現れる証明と似た記法で書くことができた．

2.4.1 Ackermann 関数とその性質

Ackermann 関数 [Ack28] は，

$$\begin{cases} ack\ 0\ y & = & \mathsf{S}\ y \\ ack\ (\mathsf{S}\ x)\ 0 & = & ack\ x\ 1 \\ ack\ (\mathsf{S}\ x)\ (\mathsf{S}\ y) & = & ack\ x\ (ack\ (\mathsf{S}\ x)\ y) \end{cases} \tag{2.5}$$

で定義される関数 $ack : \mathbf{nat} \to \mathbf{nat} \to \mathbf{nat}$ のことであり，計算可能であるが原始再帰的 (primitive recursive) でない関数の例として有名である[*6]．ack が原始再帰的でないことを示す方法はいくつかあるが，最も有名なものは，ack が任意の原始再帰的関数よりも増加が速いことをいうものであり，その中では，補題として ack が満たす以下の等式や不等式を証明することになる．

補題 1　$\forall y : \mathbf{nat},\ ack\ 1\ y = \mathsf{S}\ (\mathsf{S}\ y)$.

補題 2　$\forall x\ y : \mathbf{nat},\ \mathsf{S}\ (\mathsf{S}\ y) \leq ack\ (\mathsf{S}\ x)\ y$

補題 3　$\forall x\ y : \mathbf{nat},\ y < ack\ x\ y$

補題 4　$\forall x\ y : \mathbf{nat},\ ack\ x\ y < ack\ x\ (\mathsf{S}\ y)$

補題 5　$\forall x\ y : \mathbf{nat},\ ack\ x\ (\mathsf{S}\ y) \leq ack\ (\mathsf{S}\ x)\ y$

補題 6　$\forall x\ y : \mathbf{nat},\ ack\ x\ y < ack\ (\mathsf{S}\ x)\ y$

補題 7　$\forall c_1\ c_2 : \mathbf{nat},\ \exists c_3 : \mathbf{nat}\ ,\ \forall x : \mathbf{nat},\ ack\ c_1\ (ack\ c_2\ x) \leq ack\ c_3\ x$

われわれは，本研究で作成したタクティックライブラリで，有用な命題について可読性の高い形式証明を書くことができることを確かめるため，上に挙げた七つの等式・不等式の証明を形式化した．以下では，その結果を述べる．

2.4.2 形式化

われわれは，前節で挙げた七つの等式・不等式の形式的証明を記述するため，まずは以下のように Ackermann 関数を定義した．

```
Fixpoint ack (x y : nat) : nat :=
  match x with
  | 0 => S y
  | S x' => let fix ackx (y : nat) :=
                match y with
                | 0 => ack x' 1
                | S y' => ack x' (ackx y')
                end
            in ackx y
  end.
```

[*6] ここでいう「原始再帰的」な関数とは，一階の（すなわち高階でない）関数 $f : \mathbf{nat}^n \to \mathbf{nat}$ であって，定数関数，後者関数，射影関数から，一階の関数の関数合成および一階の関数の原始再帰法を繰り返し適用することによって得られるものを指す．厳密な定義については，例えば，有川による教科書 [有川 86] を参照．プログラミング言語分野では一般的な，高階関数の使用を許した原始再帰（primitive recursion）の定義では，ack も原始再帰ということになるので注意が必要である．

```
1  Proposition ack_1_y_eq_SSy :
2    forall (y : nat), ack 1 y = S (S y).
3  Proof.
4    induction y.
5    - @ goal : (ack 1 0 = 2).
6      Left
7      = 2.
8      = Right.
9    - @ goal : (ack 1 (S y) = S (S (S y))).
10     @ IHy  : (ack 1 y = S (S y)).
11     Left
12     = (ack 1 (S y)).
13     = (ack 0 (ack 1 y)).
14     = (ack 0 (S (S y)))      { by IHy }.
15     = (S (S (S y))).
16     = Right.
17 Qed.
```

図 2.7　補題 1（$\forall\, y\ :\mathbf{nat},\ ack\ 1\ y = S\ (S\ y)$）を証明するスクリプト

この定義には Fixpoint コマンドを使っているが，このコマンドには停止性の明らかな再帰関数しか定義できないという制約がある[*7]．その制約により，この ack の定義は，(2.5) 式で示した非形式的定義とは一見異なるものになっている．しかし，この定義は，ack の定義中で，ackx = ack (S x') なる関数を局所的に定義して，現れる ack (S x') を ackx に置き換えただけで，実際には非形式的定義と同じものである．実際，以下の三つの補題が成り立つことが容易に示せる．

- forall y, ack 0 y = S y
- forall x y, ack (S x) 0 = ack x 1
- forall x y, ack (S x) (S y) = ack x (ack (S x) y)

この定義の上で，先に挙げた七つの補題のうち，**補題 1** および**補題 2** を示したスクリプトを，それぞれ図 2.7 および図 2.8 に示す．

補題 2 は，先に挙げた七つの不等式の中で唯一，二重帰納法を用いる証明であり最も複雑なものであると考えられるが，図 2.8 のスクリプトは，以下の点のおかげで，人間が読んでも十分に理解可能なものになっている．

- 証明自体が宣言的な記法で書かれており，変形のステップを明確に理解することができる．また，アノテーションによって重要な変数とその型（今回の例であれば帰納法の仮定）が明示されており，どういう仮定を用いたのか極めて理解しやすいものになっている．
- アノテーションによって，帰納法のゴールごとにサブゴールを明示しており，人間が証明を理解する助けになっている．

また，図 2.8 のスクリプトについて，次の点を指摘しておきたい．この証明において，y についての帰納法の induction step（21 から 30 行目）では，右辺から左辺に至る方向（leftwards）

[*7] 具体的には，再帰呼び出しごとに引数のコンストラクタが「減少」するような関数しか定義できない．

```
1  Proposition ack_SSy_le_ack_Sx_y :
2    forall (x y : nat), S (S y) <= ack (S x) y.
3  Proof.
4    induction x.
5    - intros y.
6      @ goal : (S (S y) <= ack 1 y).
7      Left
8      = (S (S y)).
9      = (ack 1 y)   { by ack_1_y_eq_SSy }.
10     = Right.
11   - induction y.
12     + @ goal : (2 <= ack (S (S x)) 0).
13       @ IHx :
14         (forall y : nat, S (S y) <= ack (S x) y).
15       Right
16       =  (ack (S (S x)) 0).
17       =  (ack (S x) 1).
18       >= (S (S 1))         { by IHx }.
19       >= 2                 { omega }.
20       = Left.
21     + @ goal : (S (S (S y)) <= ack (S (S x)) (S y)).
22       @ IHx  : (forall y : nat, S (S y) <= ack (S x) y).
23       @ IHy  : (S (S y) <= ack (S (S x)) y).
24       Right
25       =  (ack (S (S x)) (S y)).
26       =  (ack (S x) (ack (S (S x)) y)).
27       >= (S (S (ack (S (S x)) y))) { by IHx }.
28       >= (S (S (S (S y))))         { by IHy }.
29       >= (S (S (S y)))             { omega }.
30       = Left.
31 Qed.
```

図 2.8　補題 2（$\forall\, x\ y : \mathbf{nat}\ ,\ \mathsf{S}\ (\mathsf{S}\ y) \leq ack\ (\mathsf{S}\ x)\ y$）を証明するスクリプト

の式変形により証明が進められているが，このことが証明全体を分かりやすいものにしていると考えられる．というのも，leftwards な変形にしたことにより，最初の変形（25 から 26 行目）が，ack の定義を用いた書き換えというごく自然なものになっているからである．もし反対に rightwards な変形で証明を書こうとしたら，最初の変形は 28 行目から 29 行目を導く（$\mathsf{S}\ (\mathsf{S}\ (\mathsf{S}\ y)) \leq \mathsf{S}\ (\mathsf{S}\ (\mathsf{S}\ (\mathsf{S}\ y)))$）という非常に「思いつきにくい」変形から始めなくてはならない．このように，leftwards な書き換えと rightwards な書き換えの両方をサポートすることは，証明の記述や理解という点において本質的なサポートとなることがある．

ここでは**補題 1** および**補題 2** に対応するスクリプトのみを掲載したが，七つのすべての等式・不等式において，こうした可読性に優れたスクリプトを得ることができた．

§2.5　プログラム運算への応用

プログラム運算においては，関係代数を用いた運算規則がいくつも知られている [Bac91, Aar92, Bir97]．それゆえ，関係代数で用いる諸規則の形式化は重要である．

```
Goal forall (R : A -|-> B) (S : B -|-> C) (T : A -|-> C),
    (S ∘ R) ∩ T ⊆ (S ∩ (T ∘ R†)) ∘ (R ∩ (S† ∘ T)).
Proof.
  intros.
  Left =
    ( S ∘ R ∩ T ).
  ⊆ { by meet_idempotent }
    ( S ∘ R ∩ (T ∩ T) ).
  ⊆ { by meet_associative }
    ( ((S ∘ R) ∩ T) ∩ T ).
  ⊆ { by modular_law_dual }
    ( (S ∩ (T ∘ R†) ∘ R) ∩ T ).  remember (S ∩ (T ∘ R†)) as U.
  ⊆ { easy }
    ( (U ∘ R) ∩ T ).
  ⊆ { by modular_law }
    ( U ∘ (R ∩ (U† ∘ T)) ).
  ⊆ { because (U ⊆ S) by ( rewrite HeqU ; apply meet_order ) }
    ( U ∘ (R ∩ (S† ∘ T)) ).
  = Right.
Qed.
```

図 2.9　モジュラ則の証明

関係代数では，例えば次の**モジュラ則**（modular law）のように，部分集合関係 $\subseteq$ を用いて表されるような規則を扱う：

$$(S \circ R) \cap T \subseteq (S \cap (T \circ R^\dagger)) \circ (R \cap (S^\dagger \circ T)). \qquad \text{(Modular-Law)}$$

集合 A から B への関係 $R \subseteq A \times B$ に対して，$R^\dagger \subseteq B \times A$ は R の逆関係と定める：

$$b\ R^\dagger\ a \iff b\ R^\dagger\ a$$

また，関係 $R \subseteq A \times B$ および $S \subseteq B \times C$ の合成 $S \circ R \subseteq A \times C$ は，

$$a\ (S \circ R)\ c \iff \exists (b \in B), (a\ R\ b \wedge b\ S\ c)$$

で定められる．

モジュラ則 (Modular-Law) は，$=$ および $\subseteq$ の鎖状記法を用いて証明することができる（詳細な証明は，例えば Bird の教科書 [Bir97] に載っている）．したがって，本章で提案したライブラリを少し改良して $=$ および $\subseteq$ の鎖状記法をサポートすることにより，図 2.9 のようなモジュラ則証明を得ることができる．同様に様々な関係代数についての諸規則の証明が形式化できると考えられ，関係代数を用いたプログラム運算への応用が期待できる．

§2.6　関連研究

本節では，本章で提案するライブラリの関連研究について述べる．我々の知る限りでは，不等式の証明や不等式変形の記述に特化したライブラリは先に例がない．しかし，宣言的証明や証明スクリプトの可読化といった観点から探してみると，いくつか関連研究を見つけることができる．

◇**宣言的証明について**　各ステップにおけるゴールを明示的に記述するような証明の記法を，宣言的記法という．定理証明支援系において宣言的記法を実現する試みはいくつかある．Isabelle/HOL をベースにした宣言的証明記述言語の設計として Isar [Bau01] が有名であるが，Coq でも Corbineau による新たな証明記述言語 C-Zar の実装 [Cor08] がある[*8]．この研究は，全体にわたって自然言語に近い形で証明を記述できるように言語をデザインするものである．そのなかには，等式変形を鎖状に近い記法で記述するためのシンタクスもあり，その点では本論文と同じアプローチによる実装が含まれていると言える．

C-Zar は，Coq をベースにしつつも新たな言語を設計しなおしていた．一方，われわれの手法では，新たな記法が Ltac の範疇で実装したタクティックライブラリとして実現されており，既存の Coq 処理系でタクティックライブラリを読み込むだけで使用できるという点に特徴を持つ．

また，C-Zar は，Coq 標準の証明に比して可読性の高さはある一方，記述量が多い上に柔軟性が低く，Coq ユーザに受け入れられなかったと言われている．山田 [山田 18] は，Coq 標準の証明から C-Zar による証明への変換器を作成し，この橋渡しを試みた．

◇**証明スクリプトの可読化ビューワについて**　本研究は，スクリプトそのものの可読性を上げようとする研究であるが，一方で，Coq 標準の証明スクリプトを，より可読性が高い形で表示する研究も盛んに行われている．Tews による Prooftree[*9]は，Coq における証明の進展をグラフィカルに表示するツールである．Kawabata ら [Kaw18] は，Prooftree を基盤として，対話的証明と連動した自然演繹風の証明木を生成するツール Traf を提案した．また，早川 [早川 18] は，現在の証明木の構造を表示するインターフェスを設計・実装した．早川らの実装では，インターフェスから証明を直接操作することも可能である．

§2.7　おわりに

本章では，Coq 上で自然数上の不等式を形式的に証明する際に，数学の教科書に現れるような記法で Coq スクリプト記述するための手法を提案した．具体的には，等式や不等式の変形をそのステップごとに変形の結果と変形ができる理由を明記しながら鎖状に記す記法や，重要な項や現在のサブゴールをアノテーションとしてスクリプト中に記す記法が，タクティックとして実現できることを示し，実際に Ltac を用いて容易に実装できることを提示した．これにより，ユーザーはタクティックライブラリを読み込むだけで，不等式変形を「鎖状」に記述することができるようになることを示した．さらに，本研究で提案するタクティックライブラリを用いて，Ackermann 関数の諸性質という情報科学において基本的な命題の証明を，スクリプトとしての可読性の高い形で記述することができることを示した．

◇**今後の課題**　今後の課題として，以下の 3 点を挙げておく．

- 本研究では自然数 **nat** 上の不等式に限定して議論したが，本研究で述べた手法の大部分は，そのまま他の型上の不等式にも拡張可能であると考えられる．しかし，自然数以外の型上の等

[*8] C-Zar は Coq 8.1 以降では標準ライブラリとして組み込まれている．また，Coq 8.3 以降は**数学的証明言語**（Mathematical Proof Language）と名称を改めている．

[*9] http://askra.de/software/prooftree/

式や不等式では，omega タクティックや ring タクティックといった強力な自動証明タクティックが動かないこともある．その場合，証明が複雑になってしまいスクリプトの記述性および可読性を下げる可能性が高い．それゆえ，自然数以外の型でも非形式的証明と同じような感覚で有用な命題を示せるようにするためには，その対象となる型ごとに特有の知見が必要になってくると考えられる．特に，実数上の不等式は応用上重要であるにも関わらず，様々な演算が入 x り組んだ複雑な式が現れるため難しい．このような，自然数以外の型への対応は重要な今後の課題である．

- 本研究では，現在のサブゴールや重要な変数の型をメモするためのアノテーションを提案した．しかし，このアノテーションは，例えば CoqIDE を用いてスクリプトを対話的に構築するといった場面では，単に冗長なだけのものである．というのも，現在のサブゴールや重要な変数の型は Goal window に表示されているのだから，わざわざスクリプト上に入力する必要がない．こうした無駄な入力を省略するために，IDE による自動補完を充実させることが重要である．こうした IDE 上の補完機能の設計は今後の課題である．
- 本研究で提案する記法によって，不等式の証明は，「自明な不等式に還元する」方針と「左辺から右辺を導く (あるいは右辺から左辺を導く)」という二つの方針が並立することになった．形式的証明において，これら二つの方針の使い分けの明確な指針があるわけではない．そうした指針の模索は今後の課題である．

第 3 章
畳み込み・反畳み込み再帰図式のための Coq ライブラリ

本章では，畳み込みおよび反畳み込み再帰図式のための Coq ライブラリについて述べる．本章の内容は，主に二つの論文 [Mur19, 村田 19b] に基づく．

§3.1 はじめに：畳み込み・反畳み込み再帰図式のための Coq ライブラリ

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った．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 [Tes11] を提案し，Bird によるレクチャノートである Theory of List [Bir87] に現れる運算規則の証明を形式化した．本章では，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す．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．また，本章では，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．

◇本章で述べられる貢献　本研究では，データ型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．データ型汎用な運算規則を形式化するアイデアは，

- 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，データ型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 [Tes11] と組み合わせ，さらに少し記法を整理することで，データ型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 [Uus99b] に現れる運算定理をすべて形式化し，Dijkstra-Feijen 流の記法を用いた可読性の高いスクリプトが得られることを確認した．

形式化の対象として選んだ Uustalu らによる論文 [Uus99b] は，histomorphism および futumorphism の提案論文である．この論文は，畳み込みおよび反畳み込み系の六つの再帰図式を含んでおり，どの再帰図式も圏 **Set** 上で定義することができるため，直観的には Coq での shallow embedding に向いていそうな再帰図式群である．しかし，

(1)　histomorphism や futumorphism のように，複雑で非自明な再帰図式を含んでおり，

(2)　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，非有界ナップ

ザック問題を解くプログラムなど，実用的にも重要な例が知られている

という二つの理由から，Uustalu らの論文が形式化できるか否かは非自明で重要な問題であると考えられる．

また，本手法では，始代数および終余代数についての型クラス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．そこで，

- histomorphism のモチベーティブな例であ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および非有界ナップザック問題を解くプログラムを，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．そこで本研究では，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た．

◇本章の構成　本章のこの後の構成は，次のようになっている．3.2 節では，圏論について簡単に復習したあと，畳み込み・反畳み込み再帰図式の理論について述べる．3.3 節では，終余代数についての型クラスのインスタンスを定義するために必要になるため，余帰納的データ型の同値性を証明するための数学的準備を行う．3.4 節では，Coq による再帰図式の形式化について述べ，始代数及び終余代数のための型クラスについて述べる．3.5 節では，型クラスのインスタンス化の例について述べ，3.6 節でその自動化について議論する．3.7 節では関連研究について述べ，3.8 節では，まとめおよび今度の課題を述べる．

§3.2　準備: 圏と再帰図式

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，詳細な解説を記した．

3.2.1　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，() は “0 項組” を表し，今後は単要素集合の要素は () で表すものとする．また，$\mathsf{inl} : X \to X + Y$ および $\mathsf{inr} : Y \to X + Y$ 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 $f : X \to A$ および $g : X \to B$ に対して，関数 $\langle f, g \rangle : X \to A \times B$ を，

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

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

を，

$$[f, g]\,(\mathsf{inl}\,a) \;=\; f\,a$$
$$[f, g]\,(\mathsf{inr}\,b) \;=\; g\,b$$

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:

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

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

二つの F 代数 (A, φ) および (B, ψ) 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，すなわち，$f \circ \varphi = \psi \circ F\,f$ となることである．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 $(\mu F, \mathsf{in}_F)$ が始 F 代数であるとは，任意の F 代数 (X, φ) 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 $(\nu F, \mathsf{out}_F)$ が終 F 余代数であるとは，任意の F 余代数 (X, φ) 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．$F\colon \mathbf{Set} \to \mathbf{Set}$ が多項式関手であるときには，必ずその始 F 代数および終 F 余代数をもつことが保証されている．始 F 代数は帰納的データ型に対応し，終 F 余代数は余帰納的データ型に対応する．

3.2.3 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，本章では特に断りがない限り圏 **Set** 上で議論する．

▶**例 3.2.1（自然数と余自然数）** 自然数関手とよばれる関手 $N\colon \mathbf{Set} \to \mathbf{Set}$ を，

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

で定める．始 N 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，終 N 余代数は，$(\mathbf{conat}, \mathit{pred})$ に対応する．ただし，**conat** は **nat** に，S が無限に続く元 $\mathsf{S}^\infty = \mathsf{S}\,(\mathsf{S}\,\cdots)$ を加えた $\mathbf{conat} = \mathbf{nat} \cup \{\mathsf{S}\,(\mathsf{S}\,(\mathsf{S}\,\cdots))\}$ であり，また $\mathit{pred}\colon \mathbf{conat} \to \mathbf{1} + \mathbf{conat}$ は，

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

である． ⊣

▶**例 3.2.2（リスト）** 集合 A に対して，リスト関手とよばれる関手 $L_A\colon \mathbf{Set} \to \mathbf{Set}$ 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．始 L_A 代数 $(\mu L_A, \mathsf{in}_{L_A})$ は $(\mathbf{list}\,A, [\boldsymbol{\lambda}\_ \Rightarrow [\,], (::)])$ に対応する．ただし，$\mathbf{list}\,A$ は A 上のリストを表す型，$[\,]$ は空リスト（nil），$(::)\colon A \to \mathbf{list}\,A \to \mathbf{list}\,A$ はリストの先頭に要素を加えるコンストラクタ（cons）である． ⊣

▶**例 3.2.3（ストリーム）** 集合 A に対して，関手 $S_A\colon \mathbf{Set} \to \mathbf{Set}$ を，

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

で定義する．始 S_A 代数は $(\mathbf{0}, \mathrm{id}_{\mathbf{0}})$ に対応し特に面白くないが，一方で終 S_A 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，$\mathbf{Stream}\,A$ 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． ⊣

3.2.4 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．

◇**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再帰図式である．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．F 代数 $\varphi\colon F\,X \to X$ に対して，φ の **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，N 代数 $[c, f] : N\,X \to X$ の 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，L_A 代数 $[c, f] : L_A\,X \to X$ の 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，各定義の第 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:

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

◇**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．F 余代数 $\varphi \colon X \to F\,X$ に対して，φ の **anamorphism** とは，図式

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

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，anamorphism $[\![\langle a, f\rangle]\!] \colon X \to \mathbf{Stream}\,A$ を考えると，$[\![\langle a, f\rangle]\!]\,x$ は，

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

の無限リストになる．

Lambeck の補題の双対により，終 F 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:

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

◇**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

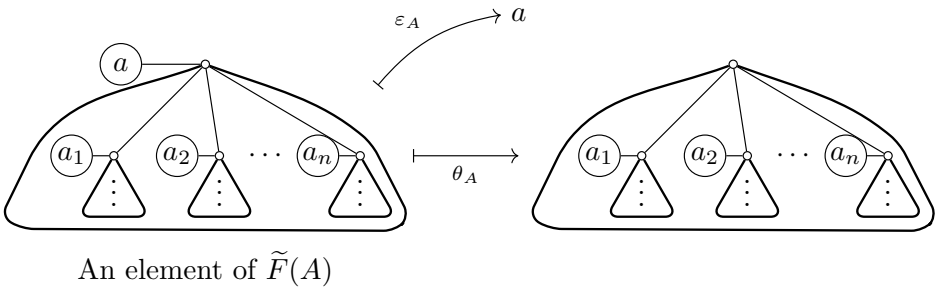


図 3.1　余自由余モナド $\widetilde{F}(A)$ の要素および，ε_A および θ_A の概念図

数を計算する関数（以下，これをフィボナッチ関数という）$fib\colon \mathbf{nat} \to \mathbf{nat}$ を考える:

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

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 $fib\,(\mathsf{S}\,(\mathsf{S}\,n))$ を計算するために，「一つ前の計算」$fib\,(\mathsf{S}\,n)$ に加え，「二つ前の計算」$fib\,n$ を呼び出しているからである．こうした計算を上手くモデリングするために，他の様々な再帰図式が考えられている．

3.2.5　Histomorphism

Histomorphism は，（直前の結果のみならず）過去のすべての再帰計算の結果を参照することができるような再帰図式である．背後には，再帰呼び出しの結果を記録しておくための余自由余モナドとよばれるメモ木を保持している．

◇**余自由余モナド**　関手 $F\colon \mathbf{Set} \to \mathbf{Set}$ に対して，$F^{\times}_{A}\colon \mathbf{Set} \to \mathbf{Set}$ を，次のように定義する:

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

この $F^{\times}_{A}$ を用いて，F の**余自由余モナド**（cofree comonad）$\widetilde{F}\colon \mathbf{Set} \to \mathbf{Set}$ が，以下のように定義される．

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

ただし，

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

直観的には $\widetilde{F}\,A$ は，νF 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 3.1 は，$\widetilde{F}\,A$ および ε_A，θ_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．ε_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，θ_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，その事実は本研究

では重要でない．

◇**Histomorphism の定義**　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，φ 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:

$$f \circ \mathsf{in}_F = \varphi \circ F\,[\![\langle f, \mathsf{in}_F^{-1}\rangle]\!]_{F_A^\times} \iff f = \{\!|\varphi|\!\}_F. \tag{3.1}$$

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 φ 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 3.1 は，この φ 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．

このような $\{\!|\varphi|\!\}_F$ は，Uustalu ら [Uus99b] により，catamorphism を用いて次のように構成できることが知られている：

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

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* は次のように定義できる：

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

3.2.6　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再帰図式として，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．

§3.3　準備: 余帰納的対象の同一性と双模倣

本節では，余帰納的に定義される対象の同一性を議論するために必要な数学的準備を行う．

ある多項式関手 F の終余代数の台 νF 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．それゆえ，例えば 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 [Chl13]．Coq の等号は，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 [San11]．

▶**定義 3.3.1**（F 双模倣） 二つの F 余代数 $\psi_0 \colon X_0 \to F\,X_0$ および $\psi_1 \colon X_1 \to F\,X_1$ に対して，関係 $R \subseteq X_0 \times X_1$ が ψ_0 と ψ_1 の F **双模倣**（F-bisilumation）であるとは，射 $e \colon R \to F\,R$ が存在して，図式

$$\begin{array}{ccccc} X_0 & \xleftarrow{\mathsf{fst}} & R & \xrightarrow{\mathsf{snd}} & X_1 \\ {\scriptstyle\psi_0}\downarrow & & \downarrow{\scriptstyle e} & & \downarrow{\scriptstyle\psi_1} \\ F\,X_0 & \xleftarrow[F\,\mathsf{fst}]{} & F\,R & \xrightarrow[F\,\mathsf{snd}]{} & F\,X_1 \end{array}$$

が可換になることである．

▶**定理 3.3.2**（双模倣による余帰納法の原理） 二項関係 $\sim_F \subseteq \nu F \times \nu F$ が F 双模倣であるとき，$\sim_F \subseteq \Delta_{\nu F} = \{(nf,\, nf) \mid nf \in \nu F\}$．

証明 $\sim_F$ は F 双模倣だから，以下の図式を可換にする $e \colon \sim_F \to F\,(\sim_F)$ が存在する．

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

$\mathsf{out}_F \colon \nu F \to F\,X_0$ が終 F 余代数であることから，$\mathsf{fst} = \mathsf{snd}$．したがって，$\sim_F \subseteq \Delta_{\nu F}$． □

したがって，νF の二つの要素 nf_0, nf_1 が等しいことを示したければ，νF の F 双模倣 $\sim_F$ に対して，$nf_0 \sim_F nf_1$ であることを示せばよい．

以下の命題は，3.5 節で 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 3.3.3** $\widetilde{N}\,\mathbf{nat}, \langle \varepsilon, \theta \rangle)$ 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，

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

が成り立つことである． ⊣

証明 素直に確かめればできる． □

§3.4 Coq による畳み込み・反畳み込み再帰図式およびその運算規則の Coq による形式化

本節では，畳み込み・反畳み込み再帰図式およびその運算規則を Coq で形式化する方法について詳しく述べる．

```
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 3.2　多項式関手の Coq による形式化

3.4.1　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．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．

図 3.2 に，多項式関手に関わる概念の Coq による定義を示す．図 3.2 の定義では，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抽象構文木の型 PolyF を定義している．PolyF のコンストラクタ zer は始対象，one は終対象，arg1 は第 1 引数の射影，arg2 は第 2 引数の射影，Sum は和型，Prod は積型を表す．例えば，

$$\texttt{Sum one (Prod arg1 arg2)} : \texttt{PolyF}$$

は，双関手 $L\,A\,X = \mathbf{1} + (A \times X)$ に対応しており，第一引数を固定して添字と見做すことによって $L_A\,X = \mathbf{1} + A \times X$ という関手であると見做せる．

型 PolyF を用いることで，F: PolyF に関する帰納的定義をすることが可能になる．先に述べた「対象から対象への関数」および「射から射への関数」も，いずれも F に関して帰納的に定義することが可能であり，図 3.2 の定義ではそれぞれ次の関数に対応する．

- 型 A に対して，inst $F\,A$: Type → Type およびその略記 $\llbracket F \rrbracket\,A$ は，型（対象）から型（対象）への関数である．
- 型 A_0, A_1 および g_a: $A_0 \to A_1$ に対して，fmap F g_a: $(X_0 \to X_1) \to \llbracket F \rrbracket\ A_0\ X_0 \to \llbracket F \rrbracket\ A_1\ X_1$ およびその略記 $F(g_a)[-]$ は関数（射）から関数（射）への関数である．

例えば，PolyF 型の項 L = Sum one (Prod arg1 arg2) を考えると，型 A に対して，$[\![L]\!]\,A$: Type → Type は $L_A\,X = \mathbf{1} + A \times X$ を表すと考えることができる．

◇**多項式関手の関手則**　多項式関手は，次の二つの関手則を満たす：

Lemma fmap_functor_dist :
∀ $(F$: PolyF$)$ $\{A_0\ A_1\ A_2\ X_0\ X_1\ X_2$: Type$\}$
$(f_0 : A_0 \to A_1)$ $(f_1 : A_1 \to A_2)$ $(g_0 : X_0 \to X_1)$ $(g_1 : X_1 \to X_2)$,
$F(f_1 \circ f_0)[g_1 \circ g_0] = F(f_1)[g_1] \circ F(f_0)[g_0]$.

Lemma fmap_functor_id :
∀ $(F$: PolyF$)$ $\{A\ X$: Type$\}$, $F(@\mathrm{id}\ A)[@\mathrm{id}\ X] = \mathrm{id}$.

この二つの命題は，F の構造に関する帰納法によって容易に証明することが可能である．なお，上の fmap_functor_id に出現する $@\mathrm{id}\,A$ は，$[\![F]\!]$ の添字 A を型推論するために必要である．

3.4.2　始代数，終余代数の形式化

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，以下の型クラス F_initial_algebra のように素直に形式化することができる．

Class F_initial_algebra $(F$: PolyF$)$ $(A$: Type$)$ $(\mu F$: Type$)$
:= {
cata : ∀ $(X$: Type$)$, $([\![F]\!]\ A\ X \to X) \to (\mu F \to X)$;
in_ : $[\![F]\!]\ A\ \mu F \to \mu F$;
cata_charn : ∀ $(X$: Type$)$ $(f : \mu F \to X)$ $(\varphi : [\![F]\!]\ A\ X \to X)$,
$f \circ$ in_ $= \varphi \circ F[f] \leftrightarrow f =$ cata $X\ \varphi$
}.
Notation "$(\!|\, f \,|\!)$" := (cata _ _ f) (at level 5).

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 の型と関数の組 $(\mu F, \mathtt{in\_})$ は，始 $[\![F]\!]\,A$ 代数を表し[*1]，cata $X\,\varphi$ は，catamorphism $(\!|\varphi|\!)_F$ を表す．また，Coq の関数 φ に対して，cata_charn は，定義した cata および in_が，本当に catamorphism および始 F 代数 in_F になっているということの証明である．

◇**Catamorphism の性質の証明**　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$)$ $(\mu F$: Type$)$ $(ia$: F_initial_algebra $F\ A\ \mu F)$

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 :
∀ $(X$: Type$)$ $(\varphi : [\![F]\!]\ A\ X \to X)$, $(\!|\varphi|\!) \circ$ in_ $= \varphi \circ F[(\!|\varphi|\!)]$.
- **Proposition** cata_refl : $(\!|$ in_ $|\!) = \mathrm{id}$.
- **Proposition** cata_fusion :
∀ $(X\ Y$: Type$)$ $(\varphi : [\![F]\!]\ A\ X \to X)$ $(\psi : [\![F]\!]\ A\ Y \to Y)$ $(f : X \to Y)$,
$f \circ \varphi = \psi \circ F[f] \to f \circ (\!|\varphi|\!) = (\!|\psi|\!)$.

[*1] 始代数の名前の末尾にアンダースコアが付されているのは，Coq の予約語である in との衝突を防ぐためである．

3.4.3 証明スクリプトの例：マップ融合則を例に

これらの命題を使って，データ型汎用なマップ融合則

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

を示してみよう．まず，型コンテキスト

Variable $(F : \mathtt{PolyF})\ (\mu F_A\ \mu F_B\ A\ B : \mathsf{Type})$
$(ia_1 : \mathtt{initial\_algebra}\ F\ A\ \mu F_A)\ (ia_2 : \mathtt{initial\_algebra}\ F\ B\ \mu F_B).$

の下で，fmap_F に相当する関数 `fmap` を

$$\textbf{Definition}\ \mathtt{fmap}\ (f : A \to B) := (\!|\ \mathtt{in\_} \circ F(f)[\mathrm{id}]\ |\!)_F$$

と定義する．この `fmap` の定義は，多項式関手 F およびインスタンス ia_1 および ia_2 の選び方によって様々なデータ型に対するマップ関数を得ることが可能な定義になっており，これによりデータ型汎用性が実現されている．この定義の下で，マップ融合則のステートメントは次のように書くことができる：

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

図 3.3 に，マップ融合則を証明する Coq スクリプトおよび，Vene [Ven00] による証明を示す．図 3.3 (a) のスクリプトと，図 3.3 (b) の証明を見比べると，かなり似た記法や方針で証明できていることがわかる．なお，Vene は，等式証明の途中で別の等式証明が必要になった場合，証明を入れ子にしたような記法を用いて対応しているが，我々は **assert** タクティックを用いて外側で示している．こうした証明の入れ子記法は，Vene（および Uustalu[*2]）独特のもので必ずしもわかりやすいと考えられるものではないため，我々のライブラリではこうした証明の入れ子記法はサポートしていない．また，Vene の証明では，我々の Coq スクリプトにおける `fmap` に対応する T をできるだけ展開せずに証明を進めているが，我々は先に **unfold** タクティックを用いて T の定義を展開したため，`fmap` の定義を展開した状態で進めている．一度展開した `fmap` をまた戻して証明を進めることもできるが，スクリプトが却って煩雑になるので，図 3.3 (a) の証明では避けている．

Tesson らの手法 [Tes11] 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，データ型汎用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．

◇提案ライブラリにおける型クラスの役割　このように手書きの証明に近い記法を実現できているのは，型クラスを用いたことによる恩恵であると言える．例えば，catamorphism $(\!|\varphi|\!)$ の定義は，多項式関手 F に依存しているが，図 3.3 のスクリプトにおいては，F そのものや始 F 代数のインスタンスを明に示すことなく証明を進めることが可能になっている．すなわち，Coq の型推論器が，F そのものや始 F 代数のインスタンスを推論することができている．このためには，Coq は，φ の型から，F および F に関連づけられた始代数と catamorphism の実装を探す必要があるが，これらが φ の型のみから推論できるのは，型クラスによるアドホック多相性を用いてい

[*2] Vene と Uustalu は多くの共著論文がある．

```
Proposition map_map_fusion :
  ∀ {A B C : Type} (f : A → B) (g : B → C), (fmap g) ∘ (fmap f) = fmap (g ∘ f).
Proof.
  intros; unfold fmap.
  assert ((fmap g) ∘ in_ ∘ F(f)[id] = in_ ∘ F(g ∘ f)[id] ∘ F(id)[fmap g]) as H0.
  {
    Left
    = ((| in_ ∘ F(g)[id] |) ∘ in_ ∘ F(f)[id]).
    = (in_ ∘ (F(g)[id] ∘ F(id)[(| in_ ∘ F(g)[id] |)]) ∘ F(f)[id])   {by cata_cancel}.
    = (in_ ∘ (F(g)[(| in_ ∘ F(g)[id] |)]) ∘ F(f)[id])               {by fmap_functor_dist}.
    = (in_ ∘ (F(g ∘ f)[(| in_ ∘ F(g)[id] |)]))                      {by fmap_functor_dist}.
    = (in_ ∘ F(g ∘ f)[id] ∘ F(id)[(| in_ ∘ F(id)[id] |)])           {by fmap_functor_dist}.
    = Right.
  }
  unfold fmap in H0.
   Left
   = ((| in_ ∘ F(g)[id] |) ∘ (| in_ ∘ F(f)[id] |)).
   = ((| in_ ∘ F(g ∘ f)[id] |))                  {apply cata_fusion}.
   Right.
Qed.
```

(a) マップ融合則を証明する Coq スクリプト（図 1.3 の再掲）

$$
\begin{array}{ll}
\left\lceil \right. & \mathsf{T}f \circ \mathsf{T}g \\
= & \quad \text{– data-map-DEF –} \\
& \mathsf{T}f \circ (\!|\, \mathrm{in} \circ \mathsf{F}(g, \mathrm{id}) \,|\!) \\
= & \quad \text{– cata-FUSION –} \\
& \quad \left\lceil \begin{array}{ll}
& \mathsf{T}f \circ \mathrm{in} \circ \mathsf{F}(g, \mathrm{id}) \\
= & \quad \text{– data-map-DEF –} \\
& (\!|\, \mathrm{in} \circ \mathsf{F}(f, \mathrm{id}) \,|\!) \circ \mathrm{in} \circ \mathsf{F}(g, \mathrm{id}) \\
= & \quad \text{– cata-SELF –} \\
& \mathrm{in} \circ \mathsf{F}(f, \mathrm{id}) \circ \mathsf{F}(\mathrm{id}, (\!|\, \mathrm{in} \circ \mathsf{F}(f, \mathrm{id}) \,|\!)) \circ \mathsf{F}(g, \mathrm{id}) \\
= & \quad \text{– F bifunctor –} \\
& \mathrm{in} \circ \mathsf{F}(f \circ g, \mathrm{id}) \circ \mathsf{F}(\mathrm{id}, (\!|\, \mathrm{in} \circ \mathsf{F}(f, \mathrm{id}) \,|\!)) \\
= & \quad \text{– data-map-DEF –} \\
& \mathrm{in} \circ \mathsf{F}(f \circ g, \mathrm{id}) \circ \mathsf{F}(\mathrm{id}, \mathsf{T}f)
\end{array} \right. \\
& (\!|\, \mathrm{in} \circ \mathsf{F}(f \circ g, \mathrm{id}) \,|\!) \\
= & \quad \text{– data-map-DEF –} \\
\left\lfloor \right. & \mathsf{T}(f \circ g)
\end{array}
$$

(b) Vene [Ven00] によるマップ融合則の証明

図 3.3　マップ融合則を証明する Coq スクリプトと Vene による証明の比較

るからである．もし型クラスを使わないと，Coq がこれらの情報を φ の型のみから推論する仕組みを用意するのは難しく，図 3.3 のスクリプトはもっと複雑なものになると考えられる．

「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．

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．

◇終余代数および anamorphism　始代数および catamorphism の場合と同様に，終余代数および anamorphism についても，次のような型クラスを用いて素直に形式化することが可能である．

$$
\begin{aligned}
&\textbf{Class}\ \texttt{F\_terminal\_coalgebra}\ (F : \texttt{PolyF})\ (A : \mathsf{Type})\ (\nu F : \mathsf{Type})\\
&\quad := \{\ \texttt{ana} \quad := \quad \forall\ (X : \mathsf{Type}), (X \to [\![F]\!]\ A\ X) \to (X \to \nu F);\\
&\qquad\ \texttt{out\_} \quad := \quad \nu F \to [\![F]\!]\ A\ \nu F;\\
&\qquad\ \texttt{ana\_charn} \ : \quad \forall\ (X : \mathsf{Type})\ (f : X \to \nu F)\ (\varphi : X \to [\![F]\!]\ A\ X),\\
&\qquad\qquad\qquad\qquad \texttt{out\_} \circ f = F[f] \circ \varphi \leftrightarrow f = \texttt{ana}\ X\ \varphi\}.\\
&\quad \}\\
&\textbf{Notation}\ \texttt{"}[\![f]\!]\texttt{"} := (\texttt{ana}\ \_\ \_\ f)\ (\text{at level } 5).
\end{aligned}
$$

3.4.4　さらに高度な運算規則の形式化

我々は，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` の定義は，

$$
\begin{aligned}
&\textbf{Definition}\ \texttt{histo}\ (F : \texttt{PolyF})\ (\mu F\ C\ \nu FC : \mathsf{Type})\\
&\qquad\qquad (ia : \texttt{F\_initial\_algebra}\ F\ C\ \mu F)\\
&\qquad\qquad (tc : \texttt{F\_terminal\_coalgebra}\ (\texttt{Prod arg1}\ F)\ C\ \nu FC)\\
&\qquad\qquad (\varphi : [\![F]\!]\ C\ \nu FC \to C)\\
&\qquad\qquad\quad := \mathsf{fst} \circ \texttt{out\_} \circ (\!|\ \texttt{out\_inv} \circ \langle \varphi, \mathrm{id} \rangle\ |\!).\\
&\textbf{Notation}\ \texttt{"}\{\!|\varphi|\!\}\texttt{"} := (\texttt{histo}\ \_\ \_\ \_\ \_\ \_\ \_\ \varphi).
\end{aligned}
\tag{3.2}
$$

のようになる．この定義において，コンストラクタ `arg1` は，余自由余モナドのための関手 $F_C^{\times}\ X = C \times F\ X$ の添字 C を表すために用いられている．

表 3.1 に，我々が Coq で定式化した再帰図式の定義（6 個）と命題（35 個）の一覧を示す．これらは，Uustalu らが論文 [Uus99b] で提案した定義および定理を全て含んでいる．

§3.5　インスタンス化

本節では，前節で述べた二つの型クラス `F_initial_algebra` および `F_terminal_coalgebra` のインスタンス化の例を示し，histomorphism を用いて定義された関数が Coq インタプリタ上で動作することを確かめる．

3.5.1　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，

$$
\textbf{Instance}\ \texttt{Nat\_ia}\ (C : \mathsf{Type}) : \texttt{F\_initial\_algebra}\ (\texttt{Sum one arg2})\ C\ \textbf{nat}
$$

を定義すれば良い．ここで，C 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 3.4 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

表 3.1 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）：誌幅の都合により型コンテキストは省略している．型コンテキストを含めた正確な定義およびステートメントは付録 A に掲載した．また，$f \otimes g$ および $f \otimes g$ はそれぞれ，射の積 $f \times g$ および余積 $f + g$ を表すが，Coq における既存の記法と衝突するため $\otimes$ および $\otimes$ を用いた．

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

```
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 3.4　nat のインスタンス化に伴う証明の例

納法によって完了する．なお，図 3.4 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．

3.5.2　終 $N^{\times}_{\mathbf{nat}}$ 余代数のインスタンス化

次に，*fib* を動かすためのメモ木のために必要な，終 $N^{\times}_{\mathbf{nat}}$ 余代数のインスタンス化を行う．まず，終 $N^{\times}_{\mathbf{nat}}$ 余代数は，Coq では次のような余帰納的データ型として定義することができる．

$$\begin{aligned}&\textbf{CoInductive}\ \texttt{mid\_tree}\\ &\quad :=\quad \mathsf{Nil}\ \ : \textbf{nat} \to \texttt{mid\_tree}\\ &\qquad\ \ |\ \mathsf{Cons} : \textbf{nat} \to \texttt{mid\_tree} \to \texttt{mid\_tree}.\end{aligned}$$

次に，終 $N^{\times}_{\mathbf{nat}}$ 余代数のための双模倣関係を，次のように定義する：

$$\begin{aligned}&\textbf{CoInductive}\ \texttt{EqMidtree}\ (t_1\ t_2 : \texttt{mid\_tree}) : \mathsf{Prop} :=\\ &|\ \texttt{eqmid}:\ \varepsilon\ t_1 = \varepsilon\ t_2\\ &\qquad\qquad \to\ \ (\theta\ t_1 = \mathsf{inl}\ () \wedge \theta\ t_2 = \mathsf{inl}\ ())\\ &\qquad\qquad\quad \vee(\forall\ a\ b, \theta\ t_1 = \mathsf{inr}\ a \to \theta\ t_2 = \mathsf{inr}\ b \to \texttt{EqMidtree}\ a\ b)\\ &\qquad\qquad \to \texttt{EqMidtree}\ t_1\ t_2.\end{aligned}$$

これが実際に双模倣関係になっていることは，命題 3.3.3 を用いれば容易に確かめられる．さて，双模倣を用いた余帰納法の原理から，$\forall\ (t_1\ t_2 : \texttt{mid\_tree}), \texttt{EqMidtree}\ t_1\ t_2 \to t_1 = t_2$ が成り立っているはずだが，これを Coq で証明することはできないことが知られている [Chl13]．それゆえ，公理として次を追加する：

$$\textbf{Axiom}\ \texttt{eq\_ext} : \forall\ (t_1\ t_2 : \texttt{mid\_tree}), \texttt{EqMidtree}\ t_1\ t_2 \to t_1 = t_2.$$

```
Definition ε (t : mid_tree) : nat :=
    match t with Nil n ⇒ n | Cons n _ ⇒ n end.
Definition θ (t : mid_tree) : nat :=
    match 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 3.5　$N^{\times}_{\mathbf{nat}}$ 終余代数のインスタンス（残った証明は省略している）

ここまでの準備のもと，終 $N^{\times}_{\mathbf{nat}}$ 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 3.5 に示す．図 3.5 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．

3.5.3　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．式 (3.2) 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，

Definition `fibo` :=
$\{\!|\, [\boldsymbol{\lambda}\_ \Rightarrow 0, [(\boldsymbol{\lambda}\_ \Rightarrow 1) \circ \mathsf{snd}, \boldsymbol{\lambda}\, p \Rightarrow (\mathsf{fst}\, p + \mathsf{snd}\, p) \circ (\mathsf{id} \otimes (\mathsf{fst} \circ \mathtt{out\_}))] \circ \mathtt{distl} \circ \mathtt{out\_} \,|\!\}$
で定義することができる．この定義の下で，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:

$$\textbf{Goal}\ (\mathtt{fibo}\, 0 = 0) \wedge (\mathtt{fibo}\, 1 = 1) \wedge \big(\forall n, \mathtt{fibo}\, (\mathsf{S}\, (\mathsf{S}\, n)) = \mathtt{fibo}\, (\mathsf{S}\, n) + \mathtt{fibo}\, n\big).$$

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．

◇非有界ナップザック問題　もう少し実用的なアルゴリズムの例として，**非有界ナップザック問題**（UKP; Unbounded Knapsack Problem）を考える．UKP は，与えられた「重さ」と「価値」のペアの品物リスト $wvs = [(w_1, v_1), \ldots, (w_n, v_n)]$ および，ナップザックの容量 c に対して，うまく自然数 $x_1, \ldots x_n \in \mathbb{N}$ を選んで，c を超えない範囲で品物の価値の総和 $\sum_{i=1}^{n} v_i x_i$ を最大化する問題である．すなわち，最適化問題風に書くと，

$$\textbf{maximize} \sum_{i=1}^{n} v_i x_i \ \textbf{subject to} \sum_{i=1}^{n} w_i x_i \leq c$$

のようになる．なお，ここでは，簡単のため，各 w_i, v_i および c は自然数型であり，さらに $w_i > 0$ であるとする．UKP は動的計画法を用いて解くことができる典型的な例として知られている．例

```
Inductive maybe {C : Type} := Nothing : maybe | Just : C → maybe.
Fixpoint midtree_nth_annotation (t : mid_tree) (n : nat) :=
  match n with
  | O    ⇒ Just (ε t)
  | S n' ⇒  match t with
            | Nil _       ⇒ Nothing
            | Cons _ t'   ⇒ midtree_nth_annotation t' n'
            end
  end.
Notation "t !! n" := (midtree_nth_annotation t n)
Fixpoint maximize {A : Type} (f : A → nat) (l : list A) :=
  match l with [] ⇒ 0 | a :: l' ⇒ max (f a) (maximize f l') end.
Definition knapsack (wvs : list (nat * nat)) :=
  {|[λ _ ⇒ 0, λ t ⇒ maximize (λ p ⇒  match p with
                                     | (w, v)  ⇒  match t !! (w − 1) with
                                                  | Nothing ⇒ 0
                                                  | Just a  ⇒ v + a
                                                  end
                                     end) wvs]|}
```

図 3.6　非有界ナップザック問題を解くプログラムの histomorphism を用いた定義

えば，次のプログラム *knapsack* は，UKP を解く Haskell 風のプログラムである：

$$
\begin{aligned}
&\mathit{knapsack}\ \mathit{wvs}\ \mathsf{O} &&= 0\\
&\mathit{knapsack}\ \mathit{wvs}\ (\mathsf{S}\ c) &&= \mathtt{maximize}\ f\ \mathit{wvs}\\
&\quad \textbf{where}\ f\ (w,\, v) = \textbf{if}\ w \leq \mathsf{S}\ c\ \textbf{then}\ v + \big(\mathit{knapsack}\ \mathit{wvs}\ (\mathsf{S}\ c - w)\big)\ \textbf{else}\ 0.
\end{aligned}
$$

ただし，`maximize` $f\ [a_1, \ldots, a_n]$ は，$f\ a_1, \ldots, f\ a_n$ のうちの最大値を返す関数である．この定義では，再帰呼び出しの *knapsack* の第二引数は $(\mathsf{S}\ c - w)$ であり，w の値が評価されるまでわからない．しかし，histomorphism では，メモ木が過去の計算結果を全て覚えているので，メモ木にアクセスすることで，第二引数を $(\mathsf{S}\ c - w)$ としたときの結果を参照することが可能である．このことに注意すると，UKP を解く histomorphism を用いた Coq プログラムは，図 3.6 のようになる．

§3.6　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` : $\forall\ f\ \varphi,\ f \circ \mathtt{in\_} = \varphi \circ F\ f \leftrightarrow f = \mathtt{cata}\ \varphi$

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 3.4 に示した．

ところで，よく考えると，三つめの証明 `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 3.4 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，μF が「綺麗に」帰納的に定義されており，`cata` および `in_` が，

```
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 3.7　タクティック auto_instance_onlyif の実装

μF の構成に沿って「素直に」定義されている場合は，三つめの証明は易しい場合も多そうである．実際，$\leftarrow$ 方向:

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

は，つまり

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

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

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

を示せば良いことになるが，これは $x\colon F\, \mu F$ 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 $\leftarrow$ 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 3.7 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 3.8 に示す五つのテストケースについて，それぞれの cata_charn の $\leftarrow$ 方向の証明が自動で完遂できるか調べた．その結果，どのテストケースにおいても，cata_charn の $\leftarrow$ 方向の証明を自動化することが可能であった．

§3.7　関連研究

本節では，本章に関わる内容の関連研究について述べる．

◇**F 代数および F 余代数，再帰図式について**　F 代数は圏論の文脈では古くから使われていた概念であり，1960 年代には，モナドや随伴とともに現れている．例えば，Lambek の補題は 1968 年の論文 [Lam68] で示されている．始 F 代数や終 F 余代数を用いたデータ型の意味論としては，Lehmann ら [Leh81] の仕事などが先駆けである．

始 F 代数から F 代数への射を catamorphism と名付け，プログラム運算の視点から捉えたのは Malcom [Mal90] である．Malcom は圏 **Set** 上で catamorphism を取り扱った（それゆえ，本

```
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 3.8 タクティック auto_instance_onlyif を評価するためのテストインスタンス

章での取り扱いに極めて近い．Catamorphism は原始再帰（primitive recursion）を全て取扱うことができなかったが，Meertens [Mee92] は原始再帰を取り扱うことができる paramorphism を提案した[*3]．その後に，Meijer ら [Mei91] は，最小元をもつ ω-cpo とその間の連続写像の成す圏 **Cppo** の上で，catamorphism, anamorphism, hylomorphism, paramorphism の 4 つの再帰図式を定義し，データ型汎用なプログラム運算の理論の礎となった．なお，hylomorphism は ω-cpo に類似の構造を用いて取り扱うのが必須なため，**Set** の範囲で取り扱える再帰図式のみを紹介した本章では述べなかったが，次章で詳しく述べる．より実用的なプログラム運算の例については，教科書 [Bir97] に catamorphism を用いたプログラム運算の例が数多く掲載されている．

1990 年代の後半，Uustalu と Vene は，新たな再帰図式を立て続けに導入する．まず，Vene ら [Ven98] は paramorphism の双対である apomorphism を導入し，さらに Uustalu ら [Uus99b] は histomorphism および futumorphism を導入する．また，Uustalu ら [Uus99a] は Mendler 流型理論の上での再帰図式を導入する．これらの成果は Vene の博士論文 [Ven00] で丁寧にまとめられており，ここまでの再帰図式の研究解説としても極めて優れている．

その後は，再帰図式を統一的に扱う研究が盛んになってくる．まず，Uustalu ら [Uus01] は，余モナドを用いて catamorphism，paramorphism，histomorphism などの畳み込み再帰図式が統一的に得られることを示した．また，Hinze ら [Hin10, Hin13] は，随伴畳み込みを用いることで様々な再帰図式を統一的に扱う手法を示している．

本章では，始代数および終余代数を用いたデータ型の取り扱いについて述べたが，圏論を用いたデータ型の取り扱いについては別の方法もある．萩野 [Hag87] は，双代数（dialgebra）の概念を用いて様々なデータ型を統一的に扱う方法を与えた．具体的には，二つの関手 $F, G\colon \boldsymbol{C} \to \boldsymbol{C}^n$ に対する F, G 双代数の圏を考え，その始対象および終対象としてデータ型および，それに付随する仲介射とよばれる関数を定義することになる．同じ萩野による CPL（Categorical Programming Language） [萩野 89] は，この洞察に基づいて設計された関数型言語である．

プログラム運算においては，寓（allegory）とよばれる関係の圏を用いたプログラム運算についての話題も重要である．寓を用いたプログラム運算は，Aarts らによる [Aar92] が先駆けであるが，教科書 [Bir97] でも丁寧に解説されており，むしろ後者の教科書が定番の文献となっている．

◇定理証明支援系を用いた運算定理の形式検証について　Mu ら [Mu09] や Chiang ら [Chi16] は，定理証明支援系 Agda を用いて，プログラム運算の教科書である Algebra of Programming [Bir97] の一部の形式化した．これらの研究は，関係の圏 **Rel** をベースにした運算定理の形式化を与えており，実際に動作するプログラムの抽出は考慮に入れていないようである．

Emoto らによる生成集約検査プログラミングの Coq による検証 [Emo14] や，Frederic らによる検証された並列プログラムを生成する Coq ライブラリ [Lou14, Lou17] は，本研究と同様にプログラム抽出を視野に入れた研究であるが，本研究とは扱っている運算定理が異なる．

[*3] [Mee92] は論文誌掲載されたのは 1992 年であるが，paramorphism 自体の発表は Meijer ら [Mei91] よりも早い．

§3.8 まとめと今後の課題

本章では，Coq により畳み込み・反畳み込み再帰図式を扱うライブラリを設計および実装し，特定の型クラスのインスタンス化を行うことで，実行可能なプログラムが得られることを確かめた．また，Uustalu らの論文 [Uus99b] に現れる定理および定義をすべて形式化し，可読性が高いスクリプトが得られることも確かめた．

今後の研究の方向性として，以下の 2 点を挙げておく．

◇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．本章で形式化された理論は 2000 年前後に提案されたものであるが，その後にはさらに高度な再帰図式の理論が提唱されている．例えば，関連研究でも挙げたように，Uustalu ら [Uus01] 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．

◇`cata_charn` の → 方向の証明の自動化　第 3.6 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．

第 4 章
Coq における Hylomorphism

本章では，Coq において hylomorphism の使いやすい形式化を得て，その上で運算を行うための基盤について述べる．

Hylomorphism を用いたプログラム運算は，適用範囲が広く有用であり，その正しさを検証するためのシステムが望まれる．定理証明支援系 Coq を用いて運算を検証するシステムはいくつか知られているが，hylomorphism およびその運算規則を使いやすい形で形式化することは困難であり，著者の知る限り未だ実現されていない．その困難さは，Coq においては帰納的データ型と余帰納的データ型が区別される一方，hylomorphism はこれらが同一視されるような条件の下で定義されるというギャップにある．本章では，このギャップを回避しつつ形式化するための技術として，遅延モナドおよび再帰的余代数の利用を検討し，例を用いてこれらの長所や短所を議論する．

§4.1 はじめに: Coq における Hylomorphism

前章では，畳み込み・反畳み込み再帰図式の形式的定義および，それらの運算規則の証明を与えた．これにより，Coq で，データ型汎用な畳み込みや反畳み込みに関わるプログラム運算を，手軽に扱うことができるようになった．一方，Meijer らによる hylomorphism [Mei91] や，Kabanov らによる dynamorphism [Kab06] をはじめとする**再畳み込み**（refolding）の再帰図式については，その有用性が知られているにも関わらず，未だ「使いやすい」形式的定義は得られていない．

本章におけるひとまずの目標は，**Coq による運算に適した実行可能な hylomorphism の定義を得る**ことである．これによって，Coq で実行可能な hylomorphism を記述し，安全に運算を行い，得られたプログラムを実行したり，他の言語に抽出して利用したりすることが可能になると考えられる．こうした目的を達成するために，前章と同様に，できるだけ shallow embedding に近い定義をしたい．では，hylomorphism が容易に Coq へ shallow embedding できるかというと，事態はそう単純ではない．というのも，一見すると矛盾しているように見える以下の二つの状況があるからである．

- **Coq で動くプログラムは全域関数である**．Coq の中核言語である Gallina は強正規化性が成り立っていて，停止するプログラムしか書くことができない．Gallina の関数 $f\colon X \to Y$ は，全域関数の圏 **Set** の射 $f\colon X \to Y$ に対応していると見做すのが自然である．
- **Hylomorphism は本質的に部分関数となり得る**．そもそも，hylomorphism は，代数コンパクト（algebrically compact） [Fre91] な関手の下でしか定義することができない．ある関手が代数コンパクトとは，端的に言えば始代数と終余代数が一致することであるが，この条件は「めったなことでは満たされない」とも言われている [長谷 03]．例えば，前章で形式化のベースとしている圏 **Set** の自己関手は，通常代数コンパクトではない．この事実は，hylomorphism

で定義された計算は，本質的に停止しない可能性を孕んでいることと深く関係している．

Meijer らの hylomorphism は，基点つき完備半順序集合（cppo）と正格連続関数の圏 $\mathbf{Cppo}_{\perp}$ の上で定義されており，$\mathbf{Cppo}_{\perp}$ の自己関手は代数コンパクトであることが知られている [Fre91]．この事実から考えれば，Coq に cppo を実装すれば良いように思えるのだが，そもそも cppo は表示的意味論に用いられる道具であり，動作可能なプログラムを得るという目的とは相性が悪いと考えられる．

全域関数しか扱えない Coq で，本質的に部分関数になり得る hylomorphism をうまく扱う方法はないものだろうか？　本章では，この問いに対する以下の二つのアプローチを検討する．

- 第一に，hylomorphism に用いる余代数を，Osius による再帰的余代数（recursive coalgebra）[Osi74] とよばれる性質の良いものに制限するアプローチを検討する．このアプローチは，大雑把に言えば，そもそも部分関数になるような「行儀の悪い」hylomorphism は扱えないように制限するという素直なものであり，理論的にも **Set** で hylomorphism を扱うために広く用いられている [Cap06, Hin15]．しかし，Coq でも容易に扱えるかどうかは定かではないため，これを確かめる．
- 第二に，Capretta による遅延モナド [Cap05] を用いるアプローチを検討する．遅延モナドは，Coq のような強正規性が成り立つ型システム上で一般再帰を実現するための手法であり，Coq でもプログラミングに用いられる [Chl13]．**Set** 上の遅延モナドが成す Kleisli 圏は，一種の部分関数の圏となるため，hylomorphism と相性が良さそうである．それどころか，遅延モナド（を弱双模倣で割ったもの）が成す Kleisli 圏は，$\mathbf{Cppo}_{\perp}$ 豊穣圏になる [Uus17] ことが知られており，$\mathbf{Cppo}_{\perp}$ に近い状況の下で hylomorphism を定義することができそうだという希望がある．これと Fokkinga [Fok94] や Pardo [Par01, Par05] らによる Kleisli 圏上での再帰図式の研究と組み合わせることで，Coq 上でうまく hylomorphism を扱えるのではないか．本章ではこのアイデアを検討する．

ところで，hylomorphism の定義のバリエーションの中には，関係の圏 **Rel** を用いたもの [Aar92, Bir97] も知られており，これを形式化することも考えられる．関係の圏は定理証明支援系との相性が良いと考えられ，実際に関係の圏をベースとして運算定理を証明した研究も存在する [Chi16, Mu09]．しかし，関係を Coq で素直に形式化すると $A \to A \to \mathsf{Prop}$ のような型となって，Gallina 処理系で動作させることのできるプログラムにはならないので，本章の目的には適さないと考えられる．

◇**本章で述べられる貢献**　本章で述べられる貢献は，以下の 2 点である．

(1)　上述の一つ目のアプローチ，すなわち再帰的余代数を用いたアプローチに検討を加えた．この方法で記述できるものは極めて再利用性の高いプログラムが得られる一方，記述しにくい例があることを指摘した．この詳細は，第 3 節で述べる．

(2)　Coq で hylomorphism を定義する第二の方法として，遅延モナドを用いたアプローチを検討した．Fokkinga [Fok94] や Pardo [Par01, Par05] による効果付き再帰図式と組み合わせて，遅延モナドを用いた遅延 hylomorphism を提案し，この定義の下で融合則などの運算法則が従うことを確かめた．また，遅延 hylomorphism の Coq での実装を行い，複数の例について動作

するプログラムが得られることを確かめた．これらについての詳細は，第 4 節で述べる．
特に (2) が主要な貢献であり，本章で提案する遅延 hylomorphism は，Coq 上で shallow embedding に近い形で hylomorphism が扱う方法として有力なものであると考えている．しかし，これらの性質についての Coq による証明は未完成であり，今後の課題となる．

◇**記法**　本章では，プログラムや命題の提示に Coq 風の記法を用いるが，可読性の向上のために，ところどころ，数学や，Haskell 等他のプログラミング言語において標準的な記法も混ぜて用いる．ただし，Coq で読み込み可能なコードに修正することが容易な範囲で濫用することにする．

◇**構成**　本章のこれ以降の構成は，次のようになる．まず，4.2 節では，hylomorphism によるプログラミングの例をいくつか挙げる．4.3 節では，Coq で hylomorphism を扱うための第一のアプローチである再帰的余代数を用いたアプローチについて検討する．4.4 節では，第二のアプローチを述べるための準備として，効果付き再帰図式および遅延モナドについて説明する．4.5 節では，Coq で hylomorphism を扱うための第二のアプローチとして，遅延モナドと効果付き再帰図式を組み合わせた遅延再帰図式を提案し，検討する．4.6 節では，本章のまとめおよび関連研究，今後の課題について述べる．

§4.2　Hylomorphism とそのプログラミング例

本節では，hylomorphism の用語を導入する．以下では，最小元を持つ ω-cpo とその間の正格な写像が成す圏を $\mathbf{Cppo}_\perp$ と書く．

4.2.1　Hylomorphism

自己関手 F および，F 余代数 $\psi\colon X \to F\,X$，F 代数 $\varphi\colon F\,Y \to Y$ に対して，$f\colon X \to Y$ の再帰方程式

$$f = \varphi \circ F\,f \circ \psi \qquad \text{(Hylo-Equation)}$$

を，**hylo 方程式**（hylo-equation）という．この Hylo-Equation は，分割統治に基づくプログラムの再帰的定義を与える上で有用である．というのも，ψ でデータを分割し，$F\,f$ による再帰で得られた結果を，φ で統治するという定義を与えられるためである．

F が多項式関手の場合は，**Set** では Hylo-Equation は解を持つとは限らないが，$\mathbf{Cppo}_\perp$ では恒に解を持つ．より一般に，F が代数コンパクト（algebrically compact）[Fre91] である場合，すなわち $\mu F = \nu F$ かつ $\mathsf{in}_F = \mathsf{out}_F^{-1}$ とできる場合には，Hylo-Equation は恒に解を持つ．具体的には，$f = (\!|\varphi|\!)_F \circ [\![\psi]\!]_F$ とすれば良い．

$\mathbf{Cppo}_\perp$ の多項式関手は，代数コンパクトであることが知られている．$\mathbf{Cppo}_\perp$ の下で，hylomorphism $[\![\varphi, \psi]\!]_F\colon X \to Y$ を，Hylo-Equation の最小解と定義する．すなわち，

$$([\![\varphi, \psi]\!]_F = \varphi \circ F\,[\![\varphi, \psi]\!]_F \circ \psi) \wedge (\forall f,\, f = \varphi \circ F\,f \circ \psi \implies [\![\varphi, \psi]\!]_F \sqsubseteq f). \qquad \text{(Hylo-Charn)}$$

この定義の下で，以下が成り立っている．

$$[\![\varphi, \psi]\!]_F = (\!|\varphi|\!)_F \circ [\![\psi]\!]_F \qquad \text{(Ana-Cata-Hylo)}$$

逆に，hylomorphism を用いて，catamorphism および anamorphism を次のように特徴付けるこ

とも可能である．

$$[\![\psi]\!]_F = [\![\mathsf{in}_F,\, \psi]\!]_F \qquad \text{(Hylo-Ana)}$$
$$(\!|\varphi|\!)_F = [\![\varphi,\, \mathsf{in}_F^{-1}]\!]_F \qquad \text{(Hylo-Cata)}$$

Hylomorphism には多くの運算法則が知られており，例えば以下の融合則が成り立つ [Fok91]．

$$h \circ \varphi = \varphi' \circ F\, h \implies h \circ [\![\varphi,\, \psi]\!]_F = [\![\varphi',\, \psi]\!]_F \qquad \text{(Hylo-FuxionC)}$$
$$\psi \circ h = F\, h \circ \psi' \implies [\![\varphi,\, \psi]\!]_F \circ h = [\![\varphi,\, \psi']\!]_F \qquad \text{(Hylo-FuxionA)}$$

4.2.2 Hylomorphism を用いたプログラミング

本章で用いる hylomorphism を用いたプログラムの例を二つ説明しておく．

◇**コラッツ列**　まず，実用性には欠けるが理論的には興味深い例として，**コラッツ列**（Collatz sequence）を取り上げる．以下，$(\%) : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ を，第一引数を第二引数で割った場合の余りを返す関数（Coq の標準ライブラリにおける `mod` に相当）とし，$(=_?) : \mathbb{N} \to \mathbb{N} \to \mathbf{bool}$ を，第一引数の自然数と第二引数の自然数が等しいか否かを返す関数（Coq の標準ライブラリにおける `=?` に相当）とする．

さて，関数 *col_succ* を，

$$\textbf{Definition}\ col\_succ\,(i : \mathbb{N}) : \mathbb{N} := \textbf{if}\ (i \,\%\, 2 =_?\, 0)\ \textbf{then}\ i/2\ \textbf{else}\ 3 * i + 1$$

で定める．正の整数 n に対して，*col_succ* の適用を繰り返してできる列

$$n,\ col\_succ\, n,\ col\_succ^2\, n,\ col\_succ^3\, n, \ldots$$

を，n を始とするコラッツ列という．例えば，3 を始とするコラッツ列は，

$$3,\ 10,\ 5,\ 16,\ 8,\ 4,\ 2,\ 1,\ 4,\ 2,\ 1,\ 4,\ 2,\ 1, \ldots \qquad (*)$$

である．この例を見てもわかるように，コラッツ列が一度 1 に至ると，そのあとはずっと 4, 2, 1, . . . の繰り返しになることから，コラッツ列の定義として 1 が出たらそこで打ち切る定義を採用する場合もある．ところで，「任意の n に対して，n を始とするコラッツ列に 1 が含まれるか？」という問題は有名な未解決問題であり，その肯定予想は**コラッツ予想**として知られている．

さて，本題に戻って，「i を始とするコラッツ列が最初に j になるのはいつか」という問題を解くための hylomorphism を定義したい．コラッツ列を生成するための余代数 $\psi_{colgen} : \mathbb{N} \to L_{\mathbb{N}}\, \mathbb{N}$ および，リストから要素 k を見つけてその番号（index）を返すための代数 $\varphi_{find}\, k : L_N\, (\mathbf{maybe}\, \mathbb{N}) \to \mathbf{maybe}\, \mathbb{N}$ を，次のように定義する：

```
Definition ψ_colgen (n : ℕ) : L_ℕ ℕ :=
  if n =? O then inl () else (if n =? 1 then inr (n, O) else inr (n, col_succ n)).
Definition φ_find (k : ℕ) (lmn : L_ℕ (maybe ℕ)) : maybe ℕ :=
  match lmn with
  | inl ()       ⇒ nothing
  | inr (n, mn)  ⇒ if n =? k then just O else (mn >>=^maybe (λ (n' : ℕ) ⇒ (just (S n'))))
  end.
```

ただし，**maybe** は Haskell 言語で一般的な maybe モナドを表し，nothing および just は maybe モナドのコンストラクタ，$\gg\!=^{\mathbf{maybe}}$ は maybe モナドの bind 演算子である．

このとき，$col\_len = [\![\varphi_{find}\, 1,\, \psi_{colgen}]\!]_F : \mathbb{N} \to \mathbf{maybe}\, \mathbb{N}$ は，入力 i を始とするコラッツ列が，

1 を出力するまでの長さを調べる関数である．ただし，例外的に $col\_len\,0 = 0$ としている．例えば，3 を始とするコラッツ列 (☒) は，先頭の 3 を 0 番目とすると，1 が初めて現れるのは 7 番目なので，$col\_len\,3 = 7$ となる．なお，この関数 col_len が全域関数になるかどうかは，明らかにコラッツ予想と同値である．したがって，$col\_len\colon \mathbb{N} \to \mathbf{maybe}\,\mathbb{N}$ が Coq で記述できるかどうかは，コラッツ予想と同程度の難問ということになる．

◇**クイックソート**　もう少し実用的な例として，クイックソートを考えてみよう．余代数 $\psi_{bstree}\colon \mathbf{list}\,\mathbb{N} \to T_{\mathbb{N}}\,(\mathbf{list}\,\mathbb{N})$ を

```
Definition ψ_bstree (ln : list ℕ) : T_ℕ (list ℕ) :=
  match ln with
  | []       ⇒ inl []
  | n :: ln' ⇒ inr (filter (λ x ⇒ x ≤? n) ln', n, filter (λ x ⇒ negb (x ≤? n)) ln')
  end.
```

で定め，代数 $\varphi_{intrav}\colon T_{\mathbb{N}}\,(\mathbf{list}\,\mathbb{N}) \to \mathbf{list}\,\mathbb{N}$ を

```
Definition φ_intrav (tln : T_ℕ (list ℕ)) : list ℕ :=
  match ln with
  | inl ()              ⇒ []
  | inr (ln'_0, n, ln'_1) ⇒ ln'_0 ++ [n] ++ ln'_1
  end.
```

で定めると，$[\![\psi_{bstree}, \varphi_{intrav}]\!]_{T_{\mathbb{N}}}\colon \mathbf{list}\,\mathbb{N} \to \mathbf{list}\,\mathbb{N}$ は，入力のリストをクイックソートする関数になる．ただし，

- $\leq_?$ は自然数の大小を bool 値で返す関数，
- *negb* は bool 値を反転させる関数，
- *filter* は第一引数で与えられた述語（**bool** 値を返す関数）を，第二引数で与えられた各要素に適用し，*true* になるものだけを残す関数，
- $+\!\!+$ はリストを結合する関数

である．

§4.3　Coq における hylomorphism (1)：再帰的余代数

本節では，Coq で hylomorphism を定義するための第一のアプローチとして，再帰的余代数 [Cap06, Hin15] を用いたものについて述べ，検討する．本節では，圏 **Set** 上で議論する．

余代数 $\psi\colon X \to F\,X$ が**再帰的**（recursive）であるとは，任意の代数 $\varphi\colon F\,Y \to Y$ に対して，

$$f = \varphi \circ F\,f \circ \psi \qquad (**)$$

を満たす f が唯一存在することである．(☒☒) は正に Hylo-Equation そのものであるから，余代数 ψ が再帰的であるとは，**Set** でも hylomorphism を考えることができるということに他ならない．この **Set** 上の hylomorhism についても，$\mathbf{Cppo}_{\perp}$ 上の hylomorphism と同様に，hylo 融合則などの多くの法則がそのまま成り立つ．

4.3.1 再帰的余代数の例と Coq による実装例

クイックソートの例で用いた $\psi_{bstree}\colon \mathbf{list}\,\mathbb{N} \to T_{\mathbb{N}}\,(\mathbf{list}\,\mathbb{N})$ は再帰的余代数である．これを示すには，(■■) を満たす f が φ に対して唯一存在することを言えば良い．Coq では，次のことを示すことになる：

Proposition bstree_reccoalg :
$\forall\,\{Y : \mathsf{Type}\}\ (\varphi\colon T_{\mathbb{N}}\,Y \to Y), \exists\,(h : \mathbf{list}\,\mathbb{N} \to Y), \forall\,(f : \mathbf{list}\,\mathbb{N} \to Y),$
$h = f \leftrightarrow f = \varphi \circ T_{\mathbb{N}}\,f \circ \psi_{bstree}$.

上の h の存在を示すため，あらかじめ次のように具体的に h を構成しておくと良い．

Function $h\ \{Y : \mathsf{Type}\}\ (\varphi : T_{\mathbb{N}}\,Y \to Y)\ (ln : \mathbf{list}\,\mathbb{N})\ \{measure\ length\ ln\} : Y :=$
match ln **with**
$\mid [\,] \Rightarrow \varphi\,(\mathsf{inl}\,())$
$\mid n :: ln' \Rightarrow \varphi\,(\mathsf{inr}\,(h\,(filter\,(\boldsymbol{\lambda}\,x \Rightarrow x \leq_?\, n)\,ln'), n, h\,(filter\,(\boldsymbol{\lambda}\,x \Rightarrow negb\,(x \leq_?\, n))\,ln')))$
end.

この **Function** コマンドは，`Coq.funind.Recdef` ライブラリに定義されているコマンドで，整礎関係上の再帰を記述するための仕組みである．上の h の場合であれば，$\{measure\ length\ ln\}$ の部分で，再帰呼び出しごとに $length\,ln$ の値が減少していくことを主張しており，この後ユーザは Coq に $length\,ln$ が本当に減少していることの証明を求められることになる．こうして無事に h が定義できれば，あとは bstree_recoalg が成り立つことを示せばよい．この bstree_recoalg の証明は難しくはないが，通常のリストに関する帰納法が使うことができず，やや面倒な証明となる．

ひとたび具体的に h を与えてしまえば，$qsort$ は $h\,\varphi_{intrav}\colon \mathbf{list}\,\mathbb{N} \to \mathbf{list}\,\mathbb{N}$ として得ることができる．

4.3.2 考察：再帰的余代数を Coq で扱うことについて

再帰的余代数によるアプローチは何ら新しいものではないが，hylomorphism によるプログラミングおよびその運算を Coq 上で支援する手法として評価しなおすとき，以下の 2 点が指摘できる．

- **停止性の証明が求められる．** クイックソートの例では，h を構成するときに，h が整礎関係上の再帰になっていることの証明，すなわち h の停止性の証明を求められた．このように，具体的な余代数が再帰的余代数であることを示すには，hylomorphism の停止性（全域性）を示す必要がある．それゆえ，コラッツ列のように，停止性の非自明な hylomorphism を扱うのは困難である（もし扱うことができるとすれば，未解決問題を解決して，コラッツ列が 1 に至ることの証明を遂行した場合のみである）．
- **停止性が証明できる余代数であっても，余代数を定義するごとに停止性の証明が求められるのは煩雑である．** 再帰的余代数を用いたアプローチでは，具体的な余代数を考えるごとに再帰的であることの証明を求められることになり，面倒である．さらに，その証明の中では，（Coq のような構成的論理の上では）具体的に hylo 方程式の解を構成することになるため，「汎用的な定義から手間をかけずに具体的なプログラムを導出する」ということができず，汎用プログラミングの技法としてはほとんど役に立たない．もちろん，hylomorphism について成り立つ運算法則を適用することで，高速なプログラムを導出することは可能であると考えられるため，

安全な運算を支援するシステムを構築するためには有用な手法であると考えられる．

§4.4　効果付き再帰図式と遅延モナド

本節では，効果付き再帰図式 [Par05] を導入し，また Capretta による遅延モナド [Cap05] について詳しく説明する．本節で提案する遅延 hylomorphism の核は，効果付き hylomorphism において，効果を与えるモナド M が遅延モナドである場合を考えることである．

4.4.1　モナドと効果付き再帰図式

モナド（monad）は，関数プログラムにおける計算効果の抽象化として重要である [Mog91]．モナドの定義は同値なものが複数知られているが，以下では join と unit を用いた (M, μ^M, η^M) によるもの，return と bind を用いた $(M, \mathsf{return}^M, (\gg\!=^M))$ によるもの，Kleisli 星を用いた $(M, \mathsf{return}^M, -^*)$ によるものなどの，すべてを用いる．混乱のおそれがない場合には，単に関手 M をモナドと見做すこともある．また，$f\colon X \to M\,Y$ および $g\colon Y \to M\,Z$ の Kleisli 合成を $g \diamond^M f\colon X \to M\,Z$ と書く．なお，ここまで導入した記法のすべてについて，M が明らかな場合には，右上の添字の M を省略することがある．なお，$g \diamond f = \mu_X \circ M\,g \circ f = \boldsymbol{\lambda} x \Rightarrow (f\,x) \gg\!= g$ である．以下では，$M\colon \boldsymbol{C} \to \boldsymbol{C}$ の Kleisli 圏を $\mathbf{Kle}_M\,\boldsymbol{C}$ で書く．混乱を避けるために，$f \in \mathrm{Hom}_{\mathbf{Kle}_M\,\boldsymbol{C}}(X, Y)$ の射を書くときには $f\colon X \to Y$ とは書かず，必ずもとの圏 $\boldsymbol{C}$ の射として $f\colon X \to M\,Y$ と書くことにする．

計算効果を伴う再帰図式として，Fokkinga [Fok94] による効果付き catamorphism や，Pardo [Par05] による効果付き anamorphism，効果付き hylomorphism が知られている[*1]．これらの基本的なアイデアは，圏 $\boldsymbol{C}$ 上の自己関手 $F\colon \boldsymbol{C} \to \boldsymbol{C}$ を Kleisli 圏に持ち上げた関手 $\widehat{F}^M\colon \mathbf{Kle}_M\,\boldsymbol{C} \to \mathbf{Kle}_M\,\boldsymbol{C}$ を定義し，この $\widehat{F}^M$ を用いて再帰図式を考えることである．

さて，関手 $\widehat{F}^M$ は，分配則（distributive law）といわれる自然変換 $\mathsf{dist}^{M,F}\colon F \circ M \Rightarrow M \circ F$ を用いて，

$$\begin{aligned} \widehat{F}^M\,X &= M\,(F\,X), \\ \widehat{F}^M\,(f\colon X \to M\,Y) &= \mathsf{dist}^{M,F}_Y \circ F\,f\colon F\,X \to M\,(F\,Y) \end{aligned}$$

と定められる．ただし，分配則は

$$\widehat{F}^M\,\mathsf{return}^M_X = \mathsf{return}^M_{F\,X}, \qquad \widehat{F}^T\,(g \diamond f) = \widehat{F}^M\,g \diamond \widehat{F}^M\,f \tag{4.1}$$

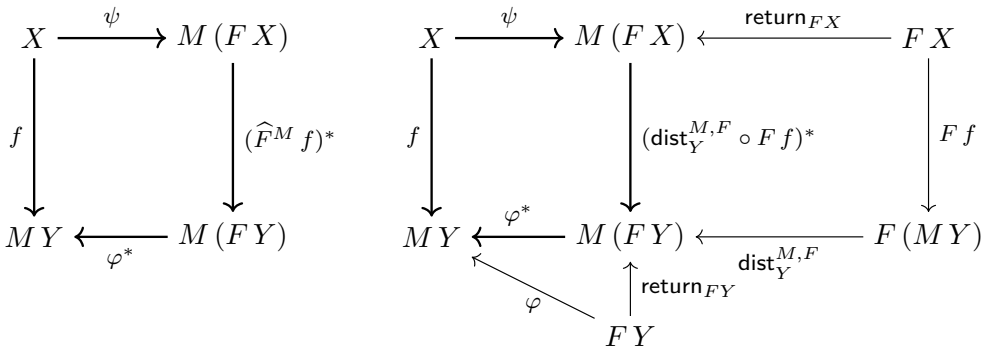
の二つを満たすように定めるとする．このおかげで，$\widehat{F}^M$ は，$\mathbf{Kle}_M\,\boldsymbol{C}$ における関手則を満たすものとなる（$\mathsf{return}^M_X \in \mathrm{Hom}_{\boldsymbol{C}}(X, M\,X)$ は，$X \in \mathrm{Obj}_{\mathbf{Kle}_M\,\boldsymbol{C}}$ の単位射であることに注意）．M が強モナド [Mog91] である場合には，(4.1) が満たされるような $\mathsf{dist}^{M,F}$ は F に関する帰納法で構成することができる．詳しくは Pardo による論文 [Par05] を参照せよ．

◇効果付き hylo 方程式　モナド M により関手 $F\colon \boldsymbol{C} \to \boldsymbol{C}$ 持ち上げた関手 $\widehat{F}^M$ を用いた，以下の効果付き hylo 方程式を考える：

$$f = \varphi \diamond \widehat{F}^M\,f \diamond \psi \tag{MHylo-Equation}$$

[*1] 「効果付き catamorphism」というような名称は本論文独自の呼称であり，元論文では monadic catamorphism などと呼んでいる．

なお，このときの各射の型は，次の二つの図式ようになっている．二つの図式はいずれも $\boldsymbol{C}$ の図式であり，右側の図式は，左側の図式の $\widehat{F}^M$ および Kleisli 星を展開したものになっている．



Pardo は，$\boldsymbol{C} = \mathbf{Cppo}$ の場合を考え，MHylo-Equation の最小解を効果付き hylomorphism $[\![\varphi, \psi]\!]_F^M$ と定めた．すなわち，

$$\begin{aligned}&([\![\varphi, \psi]\!]_F^M = \varphi \diamond \widehat{F}^M\, [\![\varphi, \psi]\!]_F^M \diamond \psi)\\ &\wedge (\forall f,\, f = \varphi \diamond \widehat{F}^M\, f \diamond \psi \implies [\![\varphi, \psi]\!]_F^M \sqsubseteq f).\end{aligned} \tag{MHylo-Charn}$$

この効果付き hylomorphism は，様々な運算法則を満たす．例えば，Hylo-FuxionC, Hylo-FuxionA と同様に，以下の融合則が成り立っている．

$$h \diamond \varphi = \varphi' \diamond \widehat{F}^M\, h \implies h \diamond [\![\varphi, \psi]\!]_F^M = [\![\varphi', \psi]\!]_F^M, \tag{MHylo-FusionC}$$

$$\psi \diamond h = \widehat{F}^M\, h \diamond \psi' \implies [\![\varphi, \psi]\!]_F^M \diamond h = [\![\varphi, \psi']\!]_F^M. \tag{MHylo-FusionA}$$

また，Hylo-Ana および Hylo-Cata と同様にして，効果付き anamorphism および効果付き catamorphism を定義することができる：

$$[\![\psi]\!]_F^M = [\![\mathsf{return}_{\mu F}^M \circ \mathsf{in}_F, \psi]\!]_F^M \qquad : X \to M\,\mu F, \tag{MHylo-MAna}$$

$$(\!|\varphi|\!)_F^M = [\![\varphi, \mathsf{return}_{F\,\mu F}^M \circ \mathsf{in}_F^{-1}]\!]_F^M \qquad : \mu F \to M\,Y. \tag{MHylo-MCata}$$

この定義の下で，anamorphism の定義, catamorphism の定義および Ana-Cata-Hylo に対応した以下が成り立っている：

$$[\![\psi]\!]_F^M \sqsubseteq f \iff f = (\mathsf{return}_{\mu F}^M \circ \mathsf{in}_F) \diamond \widehat{F}^M\, f \diamond \psi \tag{MAna-Charn}$$

$$(\!|\varphi|\!)_F^M \sqsubseteq f \iff f = \varphi \diamond \widehat{F}^M\, f \circ \mathsf{in}_F^{-1} \tag{MCata-Charn}$$

$$[\![\varphi, \psi]\!]_F^M = (\!|\varphi|\!)_F^M \diamond [\![\psi]\!]_F^M. \tag{MAna-MCata-MHylo}$$

4.4.2 遅延モナド

データ型 X に対して，余帰納的データ型 $\mathsf{D}\,X$ を，二つのコンストラクタ $\ulcorner - \urcorner$ および $\triangleright$ を用いて，以下のように定義する．

$$\textbf{CoInductive}\ \mathsf{D}\ \{X : \mathsf{Type}\} : \mathsf{Type} := \ulcorner - \urcorner : X \to \mathsf{D}\,X \mid \triangleright : \mathsf{D}\,X \to \mathsf{D}\,X.$$

あとで述べるように，D はモナドになっているので，D を**遅延モナド**（delay monad）という．

直観的には，$\ulcorner x \urcorner$ は今すぐ項 x が得られることを意味し，$\triangleright$ は項が得られるのが 1 ステップ遅延することを表す．例えば，$\triangleright\triangleright\triangleright\ulcorner \mathsf{true} \urcorner : \mathsf{D}\,\mathbf{bool}$ であれば，3 ステップ後に値 true が返されることを意味する．D は余帰納的に定義されているので，$\triangleright$ が無限に続く元 $\triangleright\ \triangleright\ \triangleright\ \cdots = \triangleright^\infty$ が存在

して，$\triangleright^\infty : \mathsf{D}\,X$ となる．Coq では，以下のように定義することができる：

$$\textbf{CoFixpoint}\ \triangleright^\infty\ \{X : \mathsf{Type}\} : \mathsf{D}\,X = \triangleright\triangleright^\infty.$$

直観的には，$\triangleright^\infty$ は，項が永久に返らないことを意味する値，すなわち無限ループに陥ることを意味する値である．

一度遅延モナドに包まれた値は，余帰納的データ型となるため，Coq では外から match 式で分解されない限り評価が進まない．したがって，遅延データ型から値を取り出すための関数として，

$$\begin{array}{l}\mathsf{Fixpoint\ evalstep}\ \{X : \mathsf{Type}\}\ (n : \mathbb{N})\ (dx : \mathsf{D}\,X) : \textbf{Maybe}\,X :=\\ \quad \textbf{match}\ (dx, n)\ \textbf{with}\\ \quad |\ (\ulcorner x \urcorner, \_) \Rightarrow \mathsf{just}\,x\\ \quad |\ (\triangleright \_, \mathsf{O}) \Rightarrow \mathsf{nothing}\\ \quad |\ (\triangleright dx', \mathsf{S}\,n') \Rightarrow \mathsf{evalstep}\ n'\ dx'\\ \quad \textbf{end}.\end{array}$$

を定義する．直観的には，$\mathsf{evalstep}\,n\,dx$ は，dx の評価を n ステップだけ進め，その間に値 $\ulcorner x \urcorner$ が得られたらその値を $\mathsf{just}\,x$ の形で返し，値が得られずに力尽きてしまったら nothing を返すものである．例えば，$\mathsf{evalstep}\,2\,(\triangleright\triangleright\ulcorner a \urcorner) = \mathsf{just}\,a$ であるが，$\mathsf{evalstep}\,1\,(\triangleright\triangleright\ulcorner a \urcorner) = \mathsf{nothing}$ である．

次に，遅延モナドに包まれた値どうしの同値性を議論するために，次の強双模倣関係 $\sim$ を考える．

$$\begin{array}{l}\textbf{CoInductive}\ (\sim)\ \{X : \mathsf{Type}\} : \mathsf{D}\,X \to \mathsf{D}\,X \to \mathsf{Prop} :=\\ \quad |\ \mathsf{sim\_value}\ : \forall\,(x : X), (\ulcorner x \urcorner \sim \ulcorner x \urcorner)\\ \quad |\ \mathsf{sim\_later}\ : \forall\,(dx_0\ dx_1 : \mathsf{D}\,X), (dx_0 \sim dx_1 \to \triangleright\,dx_0 \sim \triangleright\,dx_1).\end{array}$$

強双模倣 $\sim$ は $\mathsf{D}\,X$ の型がつく要素同士の等しさとしてかなり「自然」なものであるから，一見すると $dx_0 \sim dx_1 \to dx_0 = dx_1$ が成り立つように見えるが，Coq ではこれは示すことができないことが知られている [Chl13]．したがって，以下の仮定を追加しておく．

$$\textbf{Axiom}\ \mathsf{bisimulation\_as\_equality} : \forall\,\{X : \mathsf{Type}\}\ (dx_0\ dx_1 : \mathsf{D}\,X), (dx_0 \sim dx_1 \to dx_0 = dx_1).$$

$\mathsf{D} : \mathsf{Type} \to \mathsf{Type}$ は関手である．実際，写像 $f : X \to Y$ に対して，$\mathsf{fmap}^\mathsf{D}\,f = \mathsf{D}\,f : \mathsf{D}\,X \to \mathsf{D}\,Y$ を，以下のように定義できる．

$$\begin{array}{l}\textbf{CoFixpoint}\ \mathsf{fmap}^\mathsf{D}\ \{X\ Y : \mathsf{Type}\}\ (f : X \to Y)\ (dx : \mathsf{D}\,X) : \mathsf{D}\,Y :=\\ \quad \textbf{match}\ dx\ \textbf{with}\\ \quad |\ \ulcorner x \urcorner \Rightarrow \ulcorner f\,x \urcorner\\ \quad |\ \triangleright dx' \Rightarrow \triangleright(\mathsf{fmap}^\mathsf{D}\ f\ dx')\\ \quad \textbf{end}.\end{array}$$

この fmap^D のもとで，D は関手則を満たす（証明は簡単な余帰納法による）．さらに，D はモナドである．$\mathsf{return}^\mathsf{D}$ および $(\gg\!=^\mathsf{D})$ は，以下のように定義することができる．

$$\begin{array}{l}\textbf{Definition}\ \mathsf{return}^\mathsf{D}\ \{X : \mathsf{Type}\}\ (x : X) : \mathsf{D}\,X := \ulcorner x \urcorner.\\ \textbf{CoFixpoint}\ (\gg\!=^\mathsf{D})\ \{X\ Y : \mathsf{Type}\}\ (dx : \mathsf{D}\,X)\ (f : X \to \mathsf{D}\,Y) : \mathsf{D}\,Y\\ \quad := \ \textbf{match}\ dx\ \textbf{with}\\ \quad\quad |\ \ulcorner x \urcorner \Rightarrow f\,x\\ \quad\quad |\ \triangleright dx' \Rightarrow \triangleright(f \gg\!=^\mathsf{D} dx')\\ \quad\quad \textbf{end}.\end{array}$$

この $\mathsf{return}^\mathsf{D}$ および $(\gg\!=^\mathsf{D})$ のもとで，D はモナド則を満たす（証明は簡単な余帰納法による）．

次に，$dx : \mathsf{D}\,X$ が値 $x : X$ に収束する（converge），すなわち，有限時間内に値 x を返すことを

表す $dx \downarrow x$ および，dx が発散する（diverge），すなわち有限時間内には値を返さないことを表す $dx^{\uparrow}$ を，以下のように定義する．

$$\begin{aligned}
&\textbf{Inductive}\ (\downarrow)\ \{A : \mathsf{Type}\} : \mathsf{D}\,A \to A \to \mathsf{Prop} :=\\
&\ |\ \mathsf{converge_now}\ :\ \forall\,(a : A),\ (\ulcorner a \urcorner \downarrow a)\\
&\ |\ \mathsf{converge_later}\ :\ \forall\,(da : \mathsf{D}\,A)\ (a : A),\ (da \downarrow a \to \triangleright da \downarrow a).\\
&\textbf{CoInductive}\ (\uparrow)\ \{A : \mathsf{Type}\} : \mathsf{D}\,A \to \mathsf{Prop} :=\\
&\ |\ \mathsf{diverge}\ :\ \forall\,(da : \mathsf{D}\,A),\ da^{\uparrow} \to (\triangleright da)^{\uparrow}.
\end{aligned}$$

例えば，$\triangleright\triangleright\ulcorner \mathsf{O} \urcorner \downarrow \mathsf{O}$ であり（これは明らかである），$(\triangleright^{\infty})^{\uparrow}$ である（これは余帰納法により容易に示せる）．

次に，弱双模倣関係 $\approx$ を，$\downarrow$ を使って次のように定義する．

$$\begin{aligned}
&\textbf{CoInductive}\ (\approx)\ \{A : \mathsf{Type}\} : \mathsf{D}\,A \to \mathsf{D}\,A \to \mathsf{Prop} :=\\
&\ |\ \mathsf{wsim_value}\ :\ \forall\,(x\ y : \mathsf{D}\,A),\ (x \downarrow a \to y \downarrow a \to x \approx y)\\
&\ |\ \mathsf{wsim_later}\ :\ \forall\,(x\ y : \mathsf{D}\,A),\ (x \approx y \to \triangleright x \approx \triangleright y).
\end{aligned}$$

また，$\approx$ は外延性によって，関数の間の関係 $\approx^{ext}$ へと次のように拡張される：

$$\textbf{Definition}\ (\approx^{ext})\ \{X\ Y : \mathsf{Type}\}\ (f_0\ f_1 : X \to \mathsf{D}\,Y) := \forall\,(x : X),\ f_0\,x \approx f_1\,x.$$

最後に，次の順序関係 $\sqsubseteq$ および $\sqsubseteq^{ext}$ を定める．

$$\begin{aligned}
&\textbf{CoInductive}\ (\sqsubseteq)\ \{X : \mathsf{Type}\} : \mathsf{D}\,X \to \mathsf{D}\,X \to \mathsf{Prop} :=\\
&\ |\ \mathsf{leq_value}\ :\ \forall\,(dx_0\ dx_1 : \mathsf{D}\,X)\ (x : X),\ dx_0 \downarrow x \to dx_1 \downarrow x \to dx_0 \sqsubseteq dx_1\\
&\ |\ \mathsf{leq_blater}\ :\ \forall\,(dx_0\ dx_1 : \mathsf{D}\,X),\ dx_0 \sqsubseteq dx_1 \to \triangleright dx_0 \sqsubseteq \triangleright dx_1\\
&\ |\ \mathsf{leq_llater}\ :\ \forall\,(dx_0\ dx_1 : \mathsf{D}\,X),\ dx_0 \sqsubseteq dx_1 \to \triangleright dx_0 \sqsubseteq dx_1.\\
&\textbf{Definition}\ (\sqsubseteq^{ext})\ \{X\ Y : \mathsf{Type}\}\ (f_0\ f_1 : X \to \mathsf{D}\,Y) := \forall\,(x : X),\ f_0\,x \sqsubseteq f_1\,x.
\end{aligned}$$

直観的には，$dx_0 \sqsubseteq dx_1$ は，dx_0 は dx_1 に（無限個も含めて）$\triangleright$ をいくつか追加して得られることを表す．特に，任意の dx に対して，$\triangleright^{\infty} \sqsubseteq dx$ が成り立っている． これらの定義の下で，以下のことが成り立つ．

$$\begin{aligned}
&\textbf{Lemma}\ \mathsf{strongsim_converge}:\\
&\quad \forall\{X : \mathsf{Type}\}\ (dx_0\ dx_1 : \mathsf{D}\,X),\ dx_0 \approx dx_1 \leftrightarrow (\forall\,(x : X),\ dx_0 \downarrow x \leftrightarrow dx_1 \downarrow x).\\
&\textbf{Lemma}\ \mathsf{sqsubseteq_converge}:\\
&\quad \forall\{X : \mathsf{Type}\}\ (dx_0\ dx_1 : \mathsf{D}\,X),\ dx_0 \sqsubseteq dx_1 \leftrightarrow (\forall\,(x : X),\ dx_0 \downarrow x \to dx_1 \downarrow x).
\end{aligned}$$

◇**不動点コンビネータ**　Φ: $(X \to \mathsf{D}\,Y) \to X \to \mathsf{D}\,Y$ が finitary であるとは，任意の f: $X \to \mathsf{D}\,Y$ および x: X に対して，$\Phi\,f\,x \downarrow y$ ならば，或る $n : \mathbb{N}$ および $x_1, \ldots x_n : X$, $y_1, \ldots, y_n : Y$ が存在して，以下の 2 条件が成り立つことである．

- $\bigwedge_{i=1}^{n} f\,x_i \downarrow y_i$
- $\forall\,(g : X \to \mathsf{D}\,Y),\ (\bigwedge_{i=1}^{n} g\,x_i \downarrow y_i) \to \Phi\,g\,x \downarrow y$

直観的には，$\Phi\,f\,x$ は，有限個の入力 $x_1, \ldots, x_n$ に対する f の値にのみ出力が依存するということである．

Capretta は，遅延モナドを用いた不動点コンビネータ

$$\mathsf{fix} : \forall\{X\ Y : \mathsf{Type}\},\ ((X \to \mathsf{D}\,Y) \to X \to \mathsf{D}\,Y) \to X \to \mathsf{D}\,Y$$

の定義を与えた．この不動点コンビネータは，以下を満たすことが示されている．

Theorem fix_least_fixed_point :
$\forall\, \{X\ Y : \mathsf{Type}\}\ (\Phi : (X \to \mathsf{D}\,Y) \to X \to \mathsf{D}\,Y),$
Φ が finitary $\to \left(\begin{array}{l} \Phi\,(\mathsf{fix}\,\Phi) \approx^{ext} \mathsf{fix}\,\Phi \\ \wedge\ \forall\,(\mathit{prefix} : X \to \mathsf{D}\,Y),\ \Phi\,\mathit{prefix} \sqsubseteq \mathit{prefix} \to \mathsf{fix}\,\Phi \sqsubseteq \mathit{prefix} \end{array} \right).$

したがって，$\mathsf{fix}\,\Phi$ は（順序 $\sqsubseteq^{ext}$ の下で）Φ の最小不動点である．Capretta は，この命題（と明らかに同値な命題）の Coq による証明を与えている[*2]．以下では，この fix のことを，遅延不動点コンビネータということにする．

◇遅延モナドの分配則　正則関手 F に対して，分配則 $\mathsf{dist}^{\mathsf{D},F} : F \circ \mathsf{D} \Rightarrow \mathsf{D} \circ F$ を，F の構造に関する帰納的定義によって定義することができる [Par05]．各 $\mathsf{dist}^{\mathsf{D},F}$ は，直観的には $\triangleright$ を全て外側に追い出すような定義になる．以下の 2 つの例を具体的に与えておく．

- リスト関手 L_A および D の分配則 $\mathsf{dist}^{\mathsf{D},L_A} : L_A \circ \mathsf{D} \Rightarrow \mathsf{D} \circ L_A$

Definition $\mathsf{dist}^{\mathsf{D},L_A}\ \{A\ X : \mathsf{Type}\}\ (\mathit{ldx} : L_A\,(\mathsf{D}\,X)) : \mathsf{D}\,(L_A\,X) :=$
match ldx **with**
| $\mathsf{inl}\,()$ $\Rightarrow$ $\ulcorner \mathsf{inl}\,() \urcorner$
| $\mathsf{inr}\,(a,\ \mathit{dax})$ $\Rightarrow$ $\mathit{dax} \gg\!= (\boldsymbol{\lambda}\, ax \Rightarrow \ulcorner \mathsf{inr}\,(a,\ ax) \urcorner)$
end.

- 木関手 T_A および D の分配則 $\mathsf{dist}^{\mathsf{D},T_A} : T_A \circ \mathsf{D} \Rightarrow \mathsf{D} \circ T_A$

Definition $\mathsf{dist}^{\mathsf{D},T_A}\ \{A\ X : \mathsf{Type}\}\ (\mathit{ldx} : T_A\,(\mathsf{D}\,X)) : \mathsf{D}\,(T_A\,X) :=$
match ldx **with**
| $\mathsf{inl}\,()$ $\Rightarrow$ $\ulcorner \mathsf{inl}\,() \urcorner$
| $\mathsf{inr}\,(\mathit{dxax}_0,\ a,\ \mathit{dxax}_1)$ $\Rightarrow$ $\mathit{dxax}_0 \gg\!= (\boldsymbol{\lambda}\, xax_0 \Rightarrow \mathit{dxax}_1 \gg\!= (\boldsymbol{\lambda}\, xax_1 \Rightarrow \ulcorner \mathsf{inr}\,(xax_0,\ a,\ xax_1) \urcorner))$
end.

§4.5　Coq における hylomorphism (2)：遅延 hylomorphism

さて，遅延 hylomorphism のアイデアは，効果付き hylomorphism において，モナド M が D であった場合を考えることである．すなわち，$\psi : X \to \mathsf{D}\,(F\,X)$ および $\varphi : F\,Y \to \mathsf{D}\,Y$ に対して，$[\![\varphi, \psi]\!]^{\mathsf{D}}_F$ を考えれば良い．しかし，我々の目的は通常の hylomorphism を Coq で扱うことなので，代数および余代数は，最初から D で汚れているものではなくて，純粋なものを与えられるようにする．すなわち，$\psi : X \to F\,X$ および $\varphi : F\,Y \to Y$ に対して，

$$[\![\varphi, \psi]\!]^{\triangleright}_F = [\![\mathsf{return}^{\mathsf{D}}_{FX} \circ \varphi, \mathsf{return}^{\mathsf{D}}_{FX} \circ \psi]\!]^{\mathsf{D}}_F$$

となる $[\![-, -]\!]^{\triangleright}_F$ を，遅延 hylomorphism の定義としたい．

以下の補題は，遅延 hylomorphism の **Set** での解を与えるためのものである．

▶補題 4.5.1（遅延 hylomorphism の基本補題）　F を多項式関手とし，関数 $\Phi_F : (X \to \mathsf{D}\,Y) \to X \to \mathsf{D}\,Y$ を，

$$\Phi_F\ f = (\mathsf{return}^{\mathsf{D}}_Y \circ \varphi) \diamond \widehat{F}^{\mathsf{D}}\ f \diamond (\mathsf{return}^{\mathsf{D}}_{FX} \circ \psi)$$

で定める．MHylo-Equation において，M を遅延モナド $\mathsf{D} : \mathbf{Set} \to \mathbf{Set}$ に特化させた方程式

$$f \approx^{ext} (\mathsf{return}^{\mathsf{D}}_Y \circ \varphi) \diamond \widehat{F}^{\mathsf{D}}\ f \diamond (\mathsf{return}^{\mathsf{D}}_Y \circ \psi).$$

[*2] 証明の全体は，http://www.duplavis.com/venanzio/publications/rec_coind.v に公開されている．

は最小解 $\mathsf{fix}\,\Phi_F$ をもつ．

証明　Φ_F が finitary であることと，fix_least_fixed_point から直ちに従う．　□

したがって，上の $\mathsf{fix}\,\Phi_F$ を，**遅延 hylomorphism** といい，$[\![\varphi, \psi]\!]^{\triangleright}_F$ と書くことにする．この定義の下で，

$$\begin{aligned}&\big([\![\varphi, \psi]\!]^{\triangleright}_F \approx^{ext} (\mathsf{return}^{\mathsf{D}}_Y \circ \varphi) \diamond \widehat{F}^{\mathsf{D}}\, [\![\varphi, \psi]\!]^{\triangleright}_F \diamond (\mathsf{return}^{\mathsf{D}}_{FX} \circ \psi)\big)\\ &\wedge \big(\forall f, f \approx^{ext} (\mathsf{return}^{\mathsf{D}}_Y \circ \varphi) \diamond \widehat{F}^{\mathsf{D}}\, f \diamond (\mathsf{return}^{\mathsf{D}}_{FX} \circ \psi) \implies [\![\varphi, \psi]\!]^{\triangleright}_F \sqsubseteq^{ext} f\big).\end{aligned} \tag{DHylo-Charn}$$

が成り立つ．

◇運算法則　MHylo-FusionC および MHylo-FusionA を，D に特化させて整理することにより，以下の DHylo-FusionC および DHylo-FusionA を得る．ただし，遅延コンビネータの性質から，MHylo-FusionC および MHylo-FusionA の $=$ は $\approx^{ext}$ に変わることに注意が必要である．

$$h \circ \varphi \approx^{ext} (\mathsf{return}^{\mathsf{D}} \circ \varphi') \diamond \widehat{F}^{\mathsf{D}}\, h \implies h \diamond [\![\varphi, \psi]\!]^{\triangleright}_F \approx^{ext} [\![\varphi', \psi]\!]^{\triangleright}_F \tag{DHylo-FusionC}$$

$$(\mathsf{return}^{\mathsf{D}} \circ \psi) \diamond h \approx^{ext} \widehat{F}^{\mathsf{D}}\, h \circ \psi' \implies [\![\varphi, \psi]\!]^{\triangleright}_F \diamond h \approx^{ext} [\![\varphi, \psi']\!]^{\triangleright}_F \tag{DHylo-FusionA}$$

◇Coq による実装　遅延不動点コンビネータのおかげで，DHylo-Charn は，ほぼそのまま Coq 風の定義に書き直すことができる．

Definition $\Phi_F\ \{X\ Y : \mathsf{Type}\}\ (\varphi : F\,Y \to Y)\ (\psi : X \to F\,X)\ (f : X \to \mathsf{D}\,Y) : X \to \mathsf{D}\,Y$
$:= \mathsf{fmap}^{\mathsf{D}}\,\varphi \circ \mu^{\mathsf{D}}_{FY} \circ \mathsf{D}\,(\mathsf{dist}^{\mathsf{D},F}_Y \circ F\,f) \circ \mathsf{return}^{\mathsf{D}}_{FX} \circ \psi.$
Definition $[\![-, -]\!]^{\triangleright}_F\ \{X\ Y : \mathsf{Type}\}\ (\varphi : F\,Y \to Y)\ (\psi : X \to F\,X) := \mathsf{fix}\,(\Phi_F\,\varphi\,\psi)$

これで，動く hylomorphism の定義を得ることができた．例えば，クイックソートの場合，

Eval compute in $\mathsf{evalstep}\,5\,([\![\varphi_{bstree}, \psi_{intrav}]\!]^{\triangleright}_{T_{\mathbb{N}}}\,[4;2;3;1])$

などとすることで，実行結果として $\mathsf{just}\,[1;2;3;4]$ を得ることができる．コラッツ列の例についても，次のように動かすことができる．

Eval compute in $\mathsf{evalstep}\,10\,([\![\varphi_{find}\,1, \psi_{colgen}]\!]^{\triangleright}_{L_{\mathbb{N}}}\,3)\ (*\implies \mathsf{just}\,(\mathsf{just}\,7)\ *).$

もちろん，evalstep に渡す「燃料」が足りなければ，すなわち，評価のステップ数が足りなければ，途中で力尽きて nothing を返してしまう：

Eval compute in $\mathsf{evalstep}\,4\,([\![\varphi_{bstree}, \psi_{intrav}]\!]^{\triangleright}_{T_{\mathbb{N}}}\,[4;2;3;1])\ (*\implies \mathsf{nothing}\ *).$

したがって，評価して具体的な値を得たいと思うたびに，入力に対して十分大きな「燃料」を与える必要がある．

4.5.1　遅延 catamorphism および遅延 anamorphism

遅延 hylomorphism が得られたので，遅延 catamorphism $(\!|-|\!)^{\triangleright}_F$ と遅延 anamorphism $[\!(-)\!]^{\triangleright}_F$ を，それぞれ次のように定義することができる．

Definition $(\!|-|\!)^{\triangleright}_F\ \{Y : \mathsf{Type}\}\ (\varphi : F\,Y \to Y) : \mu F \to \mathsf{D}\,Y := [\![\varphi, \mathsf{in}^{-1}_F]\!]^{\triangleright}_F.$
Definition $[\!(-)\!]^{\triangleright}_F\ \{X : \mathsf{Type}\}\ (\psi : X \to F\,X) : X \to \mathsf{D}\,\mu F := [\![\mathsf{in}_F, \psi]\!]^{\triangleright}_F.$

注目すべきは，遅延 anamorphism の型が，$\psi : X \to F\,X$ に対して $[\![\psi]\!]^{\triangleright}_F : X \to \mathsf{D}\,\mu F$ となっており，D に包まれてはいるものの μF を返している点である．**Set** の anamorphism は νF を返す

射であったのに対し，遅延 anamorphism は μF を返す射とすることができる．これは，直観的には，無限木を成長させる anamorphism $[\![\psi]\!]_F\, x$ は，遅延 anamorphism においては $[\![\psi]\!]^{\triangleright}_F\, x = \triangleright^{\infty}$ とすることができるためである．

MAna-MCata-MHylo を D に特化することにより，以下が得られる．

$$[\![\varphi, \psi]\!]^{\triangleright}_F \approx^{ext} (\!|\varphi|\!)^{\triangleright}_F \diamond [\![\psi]\!]^{\triangleright}_F. \qquad \text{(DAna-DCata-DHylo)}$$

◇**Coq による実装**　Coq による実装は上の通りであるが，動作例も二つほど示しておく．

Eval compute in evalstep 10 ($[\![\psi_{colgen}]\!]^{\triangleright}_{L_{\mathbb{N}}}$ 3)　　(* $\Longrightarrow$ just [3; 10; 5; 16; 8; 4; 2; 1] *).
Eval compute in evalstep 10 ($(\!|\varphi_{find}\, 3|\!)^{\triangleright}_{L_{\mathbb{N}}}$ [0; 1; 2; 3; 4])　　(* $\Longrightarrow$ just 3 *).

4.5.2　考察：遅延 hylomorphism を Coq で扱うことについて

本研究では，Coq への shallow embedding ができる hylomorphism として，遅延 hylomorphism を提案した．4.3.2 節と同様に，遅延 hylomorphism を Coq による hylomorphism を用いたプログラミングおよびその運算を支援する手法として評価するとすれば，以下の 2 点が指摘できる．

- **汎用プログラミングの技法として簡便で強力である．**遅延 hylomorphism を用いることで，Coq の中に，F, φ, ψ について汎用な高階関数 $[\![\varphi, \psi]\!]^{\triangleright}_F$ を定義することができた．それゆえ，ユーザは，F, φ, ψ の定義を与えるだけで，動作するプログラムを得ることができる．特に，コラッツ列のような停止性が非自明な関数となる hylomorphism を容易に扱うことができた．したがって，プログラミングの手法としては，再帰的余代数を用いたアプローチよりかなり簡便で強力な手法であると言えそうである．
- **出力が遅延モナド型で包まれていることによる面倒やオーバーヘッドがあり得る．**遅延 hylomorphism の出力は，余帰納的なデータ型 D で包まれているため，Coq の評価器で具体的な値を取り出すには evalstep のような関数が必要である．これは再帰的余代数による hylomorphism にはなかった面倒であると言える．また，D に包まれているぶん，計算にはオーバヘッドが生じている．特に，Coq で遅延 hylomorphism を定義するために用いた fix コンビネータは，実用を目指して定義されたものというよりは，遅延モナドを用いた場合の計算可能性について議論するために理論的に導入されたものであり，その動作は実用に耐えるほど高速とは言えないと考えられる．

§4.6　まとめと今後の課題

本節では，本章のまとめと今後の課題について述べる．

◇**まとめ**　本章では，Coq で hylomorphism を扱う方法として，再帰的余代数を用いたものと，遅延 hylomorphism を用いたものの二つを検討を行った．特に後者の遅延 hylomorphism については，Coq で実行および運算が可能な hylomorphism を扱うことが可能であり，理論的な整理を行うことで，遅延 hylomorphism は通常の hylomorphism と似た運算規則を持つことも明らかにした．それゆえ，遅延 hylomorphism は，Coq で hylomorphism を扱うための有用な道具であることが示唆された．

また，遅延 hylomorphism は，理論的に興味深い遅延モナドの応用を示せたと考えている．こ

れまで，再畳み込み系の再帰図式は，主に $\mathbf{Cppo}_{\perp}$ などの代数コンパクトな圏の上で定義されるものであった．$\mathbf{Cppo}_{\perp}$ では，始対象 $\mathbf{0}$ と終対象 $\mathbf{1}$ が一致することが知られているが，型 $\mathbf{0}$ を持つプログラムが書けることになり，証明系としては矛盾を導くことになると考えられる[*3]．再畳み込み系の再帰図式を，Curry-Howard に基づく定理証明支援系である Coq で扱うことのむずかしさは，ここにある．しかし，遅延モナドの Kleisli 圏 $\mathbf{Kle}_{\mathsf{D}}\,\mathbf{Set}$ では，カルテシアン閉圏 $\mathbf{Set}$ のもつ証明系としての性質と，遅延 hylomorphism のもつ種々の性質を両立させられることになる．本章で述べた遅延再帰図式は，この両立をうまく利用することで，停止するとは限らない hylomorphism を定理証明支援系で扱う一つの方法を示唆するものであると言える．

◇**今後の課題**　今後の課題としては，以下の 4 点を挙げておく．

(1) **遅延 hylomorphism について成り立つ法則の検証**．Pardo らの効果付き hylomorphism について成り立つ性質を遅延モナド D へ特化させることによって，遅延 hylomorphism について成り立つ法則が得られるが，これらの法則の Coq による証明は未完成である．これらの証明は通常等式や不等式を用いた推論によって行われるが，これには第 2 章と同様の手法を使うことができると考えている．

(2) **遅延 fix コンビネータを用いることによる計算速度への影響の調査**．4.5.2 節でも指摘したように，Coq における遅延 hylomorphism は，遅延モナドを用いた fix コンビネータを用いて定義されているが，この fix コンビネータは実用に耐えるほど高速とは言えないと考えられる．具体的にどれぐらいのオーバヘッドが生ずるのか調査したり，より高速で簡便な代替実装を検討することは，今後の課題である．

(3) **酸性雨定理の証明**．Takano らによる酸性雨定理 [Tak95] は，hylomorphism に関わる強力な運算法則の一つであり，是非とも Coq での形式化が望まれる．酸性雨定理の証明としては free theorem [Wad89] を用いた証明が知られているため，Coq で free theorem をうまく扱う方法を模索すると良いと考えられる．この方向では，Keller による paramcoq ライブラリ [Kel12] が利用できると考えられるが，詳細な検討は今後の課題である．

(4) **遅延 Dynamorphism**．Kabanov らによる dynamorphism [Kab06] は，動的計画法を扱う再帰図式として有用なものである．本章で遅延 hylomorphism を提案したのと同様に，遅延 dynamorphism のようなものを定義すれば，Coq で動的計画法を用いたプログラミングやその運算を行うことが可能になると考えられる．このアイデアの詳細な検討は，今後の課題である．

[*3] ただし，$\mathbf{Cppo}_{\perp}$ はカルテシアン閉圏ではないので，厳密には Curry-Howard-Lambeck 対応の議論をそのまま適用できるわけではなく，完全に「矛盾」とは言い切れない．全ての対象が始対象になる圏（つまり，一点圏と圏同値な圏）を，**矛盾した圏**（inconsistent category）という．カルテシアン閉圏においては，$\mathbf{0} \cong \mathbf{1}$ ならば矛盾した圏になることが知られているが，$\mathbf{Cppo}_{\perp}$ はカルテシアン閉圏ではないため，一点圏に潰れてしまう心配はないのだが，その分証明系としての性質はよくわからないものとなっている．

第 5 章
おわりに

本章では，本論文のまとめおよび，今後の課題について述べる．

§5.1 まとめ

本論文では，Coq により再帰図式を扱うライブラリを設計および実装し，畳み込み・反畳み込み系の再帰図式を用いた運算可能なプログラムが得られることを確かめた．また，再畳み込み系の再帰図式を Coq で扱うために，遅延再帰図式を定義し，これを用いて再畳み込み計算についても運算可能なプログラムが得られることを確かめた．これによって，

- ユーザが再帰図式を用いてプログラムを作成し，
- 完成したプログラムに検証された運算規則を適用することで，高速なプログラムを安全に導出し，
- 導出されたプログラムを実行したり，必要に応じて，Coq のプログラム抽出機能を用いて他のプログラミング言語で再利用

することができるようになった．本論文で提案するライブラリは，様々なデータ型に対して汎用的な運算規則を与えているため，Coq による実用的なプログラムの運算も視野に入ってきた．

§5.2 今後の課題

既にトピックごとの細かな今後の課題については，第 2 章から第 4 章の各章末で述べた．ここでは，研究全体に関する大域的な今後の課題を二つほど示しておく．

◇計算量の保証，精度の保証　本論文は，Coq を用いた再帰図式の「正しさの保証」にのみ焦点を当てた．しかし，現実的な計算においては，計算量の保証も重要である．Coq を用いて計算量の保証付プログラムをる方法はいくつか知られており，例えば McCarthy ら [McC18] は，モナドのような構造を用いて Coq プログラムとその計算量に関する命題を結びつけることによって，Coq において計算量の保証つきプログラムを定義する方法を示した．このような手法と組み合わせることにより，プログラム運算によって本当に計算量が改善されているか否かの保証をする枠組みを設計することは，今後の重要な課題である．

さらに，浮動小数点のようなデータ型を用いる場合には，計算の精度の保証も重要になってくる．近年，精度保証つき数値計算の研究が盛んに行われているが，このような精度保証を Coq で容易に扱う仕組みを作ることによって，「精度保証つきプログラム運算」のようなことはできないだろうか．このアイデアを検討することも，今後の課題である．

◇並列計算への応用　プログラム運算定理の中には，逐次プログラムから並列プログラムを導出できるものもある．例えば，Gibbons によるリスト第三準同型定理 [Gib96] および Morihata らによるその一般の木への拡張 [Mor09] は，逐次プログラムから並列プログラムを自動導出する枠

組みとして重要である．Loulergue ら [Lou17] は，リスト準同型に基づいた並列プログラムの導出の正当性を Coq で検証し，自動導出までサポートした研究であるが，データ型汎用性は実現できていない．Morihatara らの一般化はデータ型汎用な運算定理を実現しており，実用上重要であると考えられるため，ぜひとも Coq による検証が望まれる．Morihata らの手法に基づき，様々なデータ型の並列プログラムを安全に導出する枠組みを Coq 上に構築できないだろうか．このアイデアを検討することも，今後の課題である．

謝辞

まず，指導教員の江本健斗准教授には，研究計画から論文執筆に至るまで，熱心なご指導をいただきました．ときには，夜遅くまで（そして夜が明けて再び朝陽が昇るまで）議論にお付き合い頂くこともありました．ここまで熱心にご指導いただけたことは，文字通りの意味で「有難い」ことだと思っています．深く感謝いたします（そして宴席での数々の失礼をお詫びいたします）．

八杉昌宏教授および光来健一教授には，中間発表の場を中心にして多数の有益なコメントをいただきました．特に，八杉先生には，研究生活に関わる様々な面白い雑談をしていただき，研究を進める上で大いに参考になりました．深く感謝いたします．

平成 31 年度に江本研究室を卒業した中島拓氏とは，2 年間研究室での生活を共にしました．特に本研究の初期には，研究に必要な勉強に付き合っていただいたり，研究についての有益なコメントをいただきました．特に，発表論文 [Mur19, 村田 18b, 村田 18a] については，中島拓氏に丁寧に読んでいただき，いくつもの有益なコメントをいただきました．ときには，夜遅くまで（そして夜が開けて再び朝日が昇って，それどころか再び夜が来るまで）議論にお付き合いい頂いたり，研究生活上生じる愚痴に付き合っていただいたり，研究の息抜きとしての雑談やお遊びに付き合っていただきました．中島拓氏に深く感謝いたします（そして宴席での数々の失礼をお詫びいたします）．

著者は整理整頓が苦手で，研究室の著者の机には，恒に書類や書籍が山積みとなっておりました．その一部は隣の机まで侵出することもありましたが，その隣の机の使用者である西野雄大君には，いつも寛大な言葉とともに笑って許していただきました．深く感謝いたします．

Twitter にいらっしゃる一流の研究者や技術者，勤勉な学生の皆様のツイートからは，技術的な事柄のみならず，研究や論文執筆に対する姿勢など，実に多くのことを学びました．Twitter のタイムラインを通して偶然知った論文が，本研究に大いに役立ったこともあります．Twitter および，Twitter のフォロイー・フォロワーのみなさまに，深く感謝いたします．

参考文献

[Aar92] Chritiene Aarts, Roland Backhouse, Paul Hoogendijk, Ed Voermans, and Jaap van der Woude. A relational theory of datatypes. (1992)

[Ack28] Wilhelm Ackermann. Zum hilbertschen aufbau der reellen zahlen. *Mathematische Annalen*. Vol. 99, No. 1, pp. 118–133. (1928)

[Awo10] Steve Awodey. *Category Theory (2nd edition)*. Oxford University Press. (2010)

[Bac91] Roland Backhouse, Peter Bruin, Grant Malcolm, T. S. Voermans, and Woude J. C. S. P.. Relational catamorphisms. *Constructing Programs from Specifications*. pp. 287–318. (1991)

[Bac03] Roland Backhouse and Jeremy Gibbons eds.. *Summer School on Generic Programming*. Vol. 2793 of *Lecture Notes in Computer Science*. Springer-Verlag. (2003)

[Bau01] Gertrud Bauer and Markus Wenzel. Calculational reasoning revisited an Isabelle/Isar experience. In proc. of *Theorem Proving in Higher Order Logics (TPHOLS 2011)*. pp. 75–90. (2001)

[Bir87] Richard Bird. An introduction to the theory of lists. In proc. of *Logic of Programming and Calculi of Discrete Design*. pp. 5–42. (1987)

[Bir97] Richard Bird and Oege de Moor. *Algebra of Programming*. Prentice-Hall. (1997)

[Cap05] Venanzio Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*. Vol. 1, No. 2, pp. 1–28. (2005)

[Cap06] Venanzio Capretta, Tarmo Uustalu, and Varmo Vene. Recursive coalgebras from comonads. *Information and Computation*. Vol. 204, pp. 437–468. (2006)

[Chi16] Yu-Hsi Chiang and Shin-Cheng Mu. Formal derivation of greedy algorithms from relational specifications: A tutorial. *Journal of Logical and Algebraic Methods in Programming*. Vol. 85, No. 5, Part 2, pp. 879–905. (2016)

[Chl13] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press. (2013)

[Coq19] Coq development team. *The Coq proof assistant reference manual (Version 8.10.2)*. (2019). https://github.com/coq/coq/releases/tag/V8.10.2

[Cor08] Pierre Corbineau. A declarative language for the Coq proof assistant. In proc. of *Types for Proofs and Programs (TYPES 2008)*. pp. 69–84. (2008)

[Cor09] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press. (2009)

[Dav90] Brian A. Davey and Hilary A. Priestley. *Introduction to lattices and order*. Cambridge University Press. Cambridge. (1990)

[Dij88] Edsger Dijkstra and W. H. Feijen. *A Method of Programming*. Addison-Wesley Longman Publishing. (1988)

[Emo14] Kento Emoto, Frédéric Loulergue, and Julien Tesson. A verified Generate-Test-Aggregate Coq library for parallel programs extraction. In proc. of *Interactive Theorem Proving*. pp. 258–274. (2014)

[Fok91] Maarten Fokkinga and Erik Meijer. Program calculation properties of continuous algebras. Technical report. CS-R9104, CWI, Amsterdam. (1991)

[Fok94] Maarten Fokkinga. Monadic maps and folds for arbitrary datatypes. Technical report. University of Twente. (1994)

[Fre91] Peter Freyd. Algebraically complete categories. In proc. of *Category Theory*. pp. 95–104. (1991)

[Gib96] Jeremy Gibbons. The third homomorphism theorem. *Journal of Functional Programming*. Vol. 6, No. 4, pp. 657–665. (1996)

[Gib07a] Jeremy Gibbons. Datatype-generic programming. In proc. of *Spring School on Datatype-Generic Programming*. (2007)

[Gib07b] Jeremy Gibbons, Meng Wang, and Bruno Cesar dos Santos Oliveira. Generic and indexed programming. In proc. of *Trends in Functional Programming (TFP 2007)*. (2007)

[Gon08] Georges Gonthier. Formal proof—the four-color theorem. *Notices of the American Mathematical Society*. Vol. 55, No. 11, pp. 1382–1393. (2008)

[Gri93] David Gries and Fred B. Schneider. *A Logical Approach to Discrete Math*. Springer-Verlag. (1993)

[Gun92] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. MIT Press. Cambridge, MA, USA. (1992)

[Hag87] Tatsuya Hagino. *Categorical Theoric Approach to Data Types*. PhD thesis. University of Edinburgh. (1987)

[Hin10] Ralf Hinze. Adjoint folds and unfolds: Or: Scything through the thicket of morphisms. In proc. of *the 10th International Conference on Mathematics of Program Construction (MPC 2010)*. pp. 195–228. (2010)

[Hin13] Ralf Hinze, Nicolas Wu, and Jeremy Gibbons. Unifying structured recursion schemes. In proc. of *the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP 2013)*. pp. 209–220. ACM. (2013)

[Hin15] Ralf Hinze, Nicolas Wu, and Jeremy Gibbons. Conjugate hylomorphisms – or: The mother of all structured recursion schemes. In proc. of *the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. pp. 527–538. (2015)

[Hor19] W. G. Horner and Gilbert Davies. XXI. A new method of solving numerical equations of all orders, by continuous approximation. *Philosophical Transactions of the Royal Society of London*. Vol. 109, . (1819)

[Hut16] Graham Hutton. *Programming in Haskell (2nd edition)*. Cambridge University Press. (2016). 〈**邦訳**：山本和彦（訳）．プログラミング Haskell 第 2 版．ラムダノート．(2019)〉

[Kab06] Jevgeni Kabanov and Varmo Vene. Recursion schemes for dynamic programming. In proc. of *8th International Conference on Mathematics of Program Construction*. pp. 235–252. (2006)

[Kaw18] Hideyuki Kawabata, Yuta Tanaka, Mai Kimura, and Tetsuo Hironaka. Traf: A graphical proof tree viewer cooperating with coq through proof general. In proc. of *the Programming Languages and Systems–16th Asian Symposium (APLAS 2018)*. pp. 157–165. (2018)

[Kel12] Chantal Keller and Marc Lasson. Parametricity in an impredicative sort. *Computer science logic–26th International Workshop/21st Annual Conference of the EACSL (CSL 2012)*. Vol. 16, pp. 381–395. (2012)

[Lam68] Joachim Lambek. A fixpoint theorem for complete categories. *Mathematische Zeitschrift*. Vol. 103, pp. 151–161. (1968)

[Leh81] Daniel Lehmann and Michael Smyth. Algebraic specification of data types: A synthetic approach. *Mathematical systems theory*. Vol. 14, No. 1, pp. 97–139. (1981)

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*. Vol. 52, No. 7, pp. 107–115. (2009)

[Lou14] Frédéric Loulergue and Julien Tesson. Certified parallel program calculation in Coq: A tutorial. In proc. of *International Conference on High Performance Computing and Simulation*

(HPCS 2014). (2014)

[Lou17] Frédéric Loulergue, Wadoud Bousdira, and Julien Tesson. Calculating parallel programs in Coq using list homomorphisms. *International Journal of Parallel Programming*. Vol. 45, No. 2, pp. 300–319. (2017)

[Mal90] Grant Malcom. Data structures and program transformation. *Science of Computer Programming*. Vol. 14, No. 2, pp. 255–279. (1990)

[McC18] Jay McCarthy, Burke Fetscher, Max S. New, Daniel Feltey, and Robert Bruce Findler. A Coq library for internal verification of running-times. *Science of Computer Programming*. Vol. 164, pp. 49–65. (2018)

[Mee92] Lambert Meertens. Paramorphisms. *Formal Aspects of Computing*. Vol. 4, pp. 413–424. (1992)

[Mei91] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In proc. of *the 5th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA 1991)*. pp. 124–144. Springer-Verlag. (1991)

[Mit96] John C. Mitchell. *Foundations of Programming Languages*. MIT Press. (1996)

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*. Vol. 93, No. 1, pp. 55–92. (1991)

[Mor09] Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. The third homomorphism theorem on trees: Downward and upward lead to divide-and-conquer. In proc. of *the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2009)*. Vol. 44. pp. 177–185. (2009)

[Mu09] Shin-Cheng Mu, Hsiang-Shang Ko, and Patrik Jansson. Algebra of programming in Agda: Dependent types for relational program derivation. *Journal of Functional Programming*. Vol. 19, No. 5, pp. 545–579. (2009)

[Mur19] Kosuke Murata and Kento Emoto. Recursion schemes in Coq. In proc. of *Programming Languages and Systems–17th Asian Symposium (APLAS 2019)*. pp. 202–221. (2019)

[Oka97] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press. (1997).〈邦訳：稲葉一浩，遠藤侑介（訳）．純粋関数型データ構造. KADOKAWA．（2017）〉

[Osi74] Gerhard Osius. Categorical set theory: A characterization of the category of sets. *Journal of Pure and Applied Algebra*. Vol. 4, No. 1, pp. 79–119. (1974)

[Par01] Alberto Pardo. Fusion of recursive programs with computational effects. *Theorical Computer Science*. Vol. 260, pp. 165–207. (2001)

[Par05] Alberto Pardo. Combining datatypes and effects. In proc. of *Advanced Functional Programming*. pp. 171–209. (2005)

[Pie91] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. MIT Press. (1991)

[San11] Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press. (2011)

[Soz10] Matthieu Sozeau. A new look at generalized rewriting in type theory. *Journal of Formalized Reasoning*. Vol. 2, No. 1, pp. 41–62. (2010)

[Sto94] V. Stoltenberg-Hansen, I. Lindström, and E. R. Griffor. *Mathematical Theory of Domains*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press. (1994)

[Tak95] Akihiko Takano and Erik Meijer. Shortcut deforestation in calculational form. In proc. of *the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA 1995)*. pp. 306–313. ACM. (1995)

[Tes11] Julien Tesson, Hideki Hashimoto, Zhenjiang Hu, Frédéric Loulergue, and Masato Takeichi.

Program Calculation in Coq. In proc. of *the 13th International Conference on Algebraic Methodology and Software Technology (AMAST 2010)*. pp. 163–179. (2011)

[Uus99a] Tarmo Uustalu and Varmo Vene. Mendler-style inductive types, categorically. *Nordic Journal of Computing*. Vol. 6, No. 3, pp. 343–361. (1999)

[Uus99b] Tarmo Uustalu and Varmo Vene. Primitive (co)recursion and course-of-value (co)iteration, categorically. *Informatica*. Vol. 10, pp. 5–26. (1999)

[Uus01] Tarmo Uustalu, Varmo Vene, and Alberto Pardo. Recursion schemes from comonads. *Nordic Journal of Computing*. Vol. 8, No. 3, pp. 366–390. (2001)

[Uus17] Tarmo Uustalu and Niccolò Veltri. The delay monad and restriction categories. In proc. of *International Conference of Theoretical Aspects of Computing (ICTAC 2017)*. pp. 32–50. (2017)

[Ven98] Varmo Vene and Tarmo Uustalu. Functional programming with apomorphisms (corecursion). In proc. of *the Estonian Academy of Sciences: Physics, Mathematics*. pp. 147–161. (1998)

[Ven00] Varmo Vene. *Categorical Programming with inductive and coinductive types*. PhD thesis. University of Tartuensis. (2000)

[Vos95] Tanja Vos. Program construction and generation based on recursive types. MSSc thesis, Department of Computer Science, University of Utrecht. (1995)

[Wad89] Philip Wadler. Theorems for free!. In proc. of *the Fourth International Conference on Functional Programming Languages and Computer Architecture (FPCA 1989)*. pp. 347–359. (1989)

[横内 94] 横内寛文．**プログラム意味論**．情報数学講座，第 10 巻．共立出版．(1994)

[嘉田 08] 嘉田勝．**論理と集合から始める数学の基礎**．日本評論社．(2008)

[山田 18] 山田伊織．定理証明支援系 Coq における手続き的証明から宣言的証明への変換．電気通信大学，情報理工学研究科，修士論文．(2018)

[早川 18] 早川恵太．定理証明支援系 Coq における証明木を操作可能なインタフェースの設計および実装．電気通信大学，情報理工学研究科，修士論文．(2018)

[村田 18a] 村田康佑，江本健斗．Coq における検証されたプログラム運算の拡張．日本ソフトウェア科学会第 35 回大会，大阪．(2018)

[村田 18b] 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）**．Vol. 11, No. 4, pp. 1–12．(2018)

[村田 19a] 村田康佑，江本健斗．Coq を用いた高度なプログラム運算定理の検証に向けて．第 21 回プログラミングおよびプログラミング言語ワークショップ（PPL2019），花巻．(2019)

[村田 19b] 村田康佑，江本健斗．高度な運算定理の Coq による証明とその自動化．日本ソフトウェア科学会第 36 回大会，東京．(2019)

[長谷 03] 長谷川立．パラメトリック・ポリモルフィズム．**コンピュータソフトウェア**．Vol. 20, No. 2, pp. 175–194．(2003)

[萩原 18] 萩原学，アフェルト・レナルド．Coq/SSReflect/MathComp **による定理証明:フリーソフトで始める数学の形式化**．森北出版．(2018)

[萩野 89] 萩野達也．カテゴリー理論的関数型プログラミング言語．**コンピュータソフトウェア**．Vol. 7, No. 1, pp. 16–32．(1989)

[有川 86] 有川節夫．**オートマトンと計算可能性**．情報処理シリーズ，第 12 巻．培風館．(1986)

付録 A
第 3 章で形式的証明を得た定理および命題一覧

本付録では，表 3.1 で述べた定理のステートメントを正確に述べるため，表 3.1 に関わる Coq のスクリプトを掲載する．ただし，証明はすべて省略する．証明つきのコードは，GitHub レポジトリ https://github.com/muratak17/Recursion-Schemes-in-Coq に掲載している．

```
(* Typeclass for F-initial algebra *)
Class F_initial_algebra (F : PolyF) (A : Type) (μF : Type) :=
  {
    cata : forall (X : Type), (⟦ F ⟧ A X -> X) -> (μF -> X) ;
    in_  : ⟦ F ⟧ A μF -> μF ;
    cata_charn : forall (X : Type) (f : μF -> X) (φ : ⟦ F ⟧ A X -> X),
        f ∘ in_ = φ ∘ F[f] <-> f = cata X φ
  }.

Notation ''(| f |)'' := (cata _ f) (at level 5).

Section catamorphism.
  Variable (F : PolyF) (A : Type) (μF : Type) (ia : F_initial_algebra F A μF).

  Proposition cata_cancel :
    forall (X : Type) (φ : ⟦ F ⟧ A X -> X), (| φ |) ∘ in_ = φ ∘ F[(| φ |)].

  Proposition cata_refl : (| in_ |) = id.

  Proposition cata_fusion :
    forall (X Y : Type) (φ : ⟦ F ⟧ A X -> X) (ψ : ⟦ F ⟧ A Y -> Y) f,
      f ∘ φ = ψ ∘ F[f] -> f ∘ (|φ|) = (|ψ|).

  Proposition lemma1 : in_ ∘ (| F[in_] |) = id /\ (| F[in_] |) ∘ in_ = id.

  Definition in_inv {F : PolyF} {A : Type} {μF : Type}
    {ia : F_initial_algebra F A μF}
    := (| F(@id A)[in_] |).

  Proposition in_inv_charn : in_ ∘ in_inv = id /\ in_inv ∘ in_ = id.
End catamorphism.

(* Terminal-coalgebra and anamorphism *)
Class F_terminal_coalgebra (F : PolyF) (A : Type) (νF : Type) :=
  {
    ana  : forall (X : Type), (X -> ⟦ F ⟧ A X) -> (X -> νF);
```

```
    out_ : νF -> ⟦ F ⟧ A νF;
    ana_charn : forall (X : Type) (f : X -> νF) (φ : X -> ⟦ F ⟧ A X),
        out_ ∘ f = F[f] ∘ φ <-> f = ana X φ
  }.

Notation ''⦇ f ⦈'' := (ana _  f) (at level 5).

Section anamorphism.
  Variable (F : PolyF) (A : Type) (νF : Type) (tc : F_terminal_coalgebra F A νF).

  Proposition ana_charn_right:
    forall (X : Type) (f : X -> νF) (φ : X -> ⟦ F ⟧ A X),
      out_ ∘ f = F[f] ∘ φ -> f =⦇ φ ⦈.

  Proposition ana_cancel:
    forall (X : Type) (φ : X -> ⟦ F ⟧ A X),
      out_ ∘⦇ φ ⦈ = F[⦇ φ ⦈] ∘ φ.

  Proposition ana_refl:⦇ out_ ⦈ = id.

  Proposition ana_fusion:
    forall (X Y : Type) (φ : X -> ⟦ F ⟧ A X) (ψ: Y -> ⟦ F ⟧ A Y) (f : X -> Y),
      ψ ∘ f = F[f] ∘ φ -> ⦇ ψ ⦈∘ f = ⦇ φ ⦈.

  Proposition out_inv_charn1_in: ⦇ F[out_] ⦈∘ out_ = id.

  Proposition out_inv_charn2_in: out_ ∘⦇ F[out_] ⦈= id.

  Definition out_inv {F : PolyF} {A : Type} {νF : Type}
             {tc : F_terminal_coalgebra F A νF}
    := ⦇ F[out_] ⦈.

  Proposition out_inv_charn1: out_inv ∘ out_ = id.

  Proposition out_inv_charn2: out_ ∘ out_inv = id.

  Proposition out_inv_charn: out_inv ∘ out_ = id /\ out_ ∘ out_inv = id.
End anamorphism.

Proposition lemma2 :
  forall (F : PolyF) (A : Type) (μF : Type)
         (ia : F_initial_algebra F A μF)
         {C : Type} (f : μF -> C) (φ : ⟦ F ⟧ A (C * μF)%type -> C),
    f ∘ in_ = φ ∘ F [⟨ f , id ⟩ ] <-> f = fst ∘ (|⟨ φ , in_ ∘ F[snd] ⟩ |).

(* Definition of paramorphisms *)
```

```
Definition para (F : PolyF) (A : Type) (μF : Type)
           (ia : F_initial_algebra F A μF)
           {C : Type} (φ : ⟦ F ⟧ A (C * μF)%type -> C)
  := fst ∘ (|⟨ φ, in_ ∘ F[snd] ⟩ |).
Notation ''(| f |)'' := (para _ _ _ _ f) (at level 5).

(* Properties of paramorphisms *)
Section paramorphism.
  Variable (F : PolyF) (A : Type) (μF : Type)
           (ia : F_initial_algebra F A μF).

  Proposition para_charn:
    forall {C : Type} (f : μF -> C) (φ : ⟦ F ⟧ A (C * μF)%type -> C),
      f ∘ in_ = φ ∘ F[⟨f, id⟩] <-> f = (|φ|).

  Proposition para_cancel:
    forall {C : Type} (φ : ⟦ F ⟧ A (C * μF)%type -> C),
      (| φ |) ∘ in_ = φ ∘ F[⟨(|φ|), id⟩].

  Proposition para_refl: id = (| in_ ∘ F[fst] |).

  Proposition para_fusion:
    forall {C D : Type} (f : C -> D) (φ : ⟦ F ⟧ A (C * μF)%type -> C)
           (ψ : ⟦ F ⟧ A (D * μF)%type -> D),
      f ∘ φ = ψ ∘ F[f ⊗ id] -> f ∘ (| φ |) = (| ψ |).

  Proposition para_cata:
    forall {C : Type} (φ : ⟦ F ⟧ A C -> C),
      (| φ |) = (| φ ∘ F[fst] |).

  Proposition para_any:
    forall {C : Type} (f : μF -> C),
      f = (| f ∘ in_ ∘ F[snd] |).

  Proposition para_in_inv: in_inv = (| F[snd] |).
End paramorphism.

(* Definition of apomorphisms *)
Definition apo (F : PolyF) (A : Type) (νF : Type)
           (tc : F_terminal_coalgebra F A νF)
           {C : Type} (φ : C -> ⟦ F ⟧ A (C + νF)%type)
  := (|[φ, F[inr] ∘ out_]|)∘ inl.

Notation ''(| f |)'' := (apo _ _ _ _ f) (at level 5).

Section apomorphism.
```

```
  Variable (F : PolyF) (A : Type) (νF : Type)
           (tc : F_terminal_coalgebra F A νF).

  Proposition apo_charn:
    forall (C : Type) (φ : C -> ⟦ F ⟧ A (C + νF)%type) (f : C -> νF),
      out_ ∘ f = F[[f, id]] ∘ φ <-> f =⟦⦇ φ ⦈⟧.

  Proposition apo_charn_onlyif :
     forall (C : Type) (φ : C -> ⟦ F ⟧ A (C + νF)%type) (f : C -> νF),
       out_ ∘ f = F[[f, id]] ∘ φ -> f =⟦⦇ φ ⦈⟧.

  Proposition apo_cancel:
     forall (C : Type) (φ : C -> ⟦ F ⟧ A (C + νF)%type),
       out_ ∘⟦⦇ φ ⦈⟧ = F[[⟦⦇ φ ⦈⟧, id]] ∘ φ.

  Proposition apo_refl: id =⟦⦇ F[inl] ∘ out_ ⦈⟧.

  Proposition apo_fusion:
    forall {C D : Type} (f : C -> D)
           (φ : C -> ⟦ F ⟧ A (C + νF)%type) (ψ : D -> ⟦ F ⟧ A (D + νF)%type),
      ψ ∘ f = F[(f ⊗ id)] ∘ φ -> ⟦⦇ ψ ⦈⟧ ∘ f =⟦⦇ φ ⦈⟧.

  Proposition apo_ana:
    forall {C : Type} (φ : C -> ⟦ F ⟧ A C),
      ⟦⦇ φ ⦈⟧=⟦⦇ F[inl] ∘ φ ⦈⟧.

  Proposition apo_any:
    forall {C : Type} (f : C -> νF),
      f =⟦⦇ F[inr] ∘ out_ ∘ f ⦈⟧.
End apomorphism.

Lemma lemma3:
  forall (F : PolyF) (μF : Type) (C : Type) (νFC : Type)
         (ia : F_initial_algebra F _ μF)
         (tc : F_terminal_coalgebra (Prod arg1 F) _ νFC),
  forall (f : μF -> C) (φ : ⟦F⟧ C νFC -> C),
    f ∘ in_ = φ ∘ F[⟦⦇ f, in_inv ⟩ ⦈⟧
    <-> f = fst ∘ out_ ∘ (| out_inv ∘⟨ φ , id ⟩ |).

Definition histo (F : PolyF) (μF : Type) (C : Type) (νFC : Type)
         (ia : F_initial_algebra F C μF)
         (tc : F_terminal_coalgebra (Prod arg1 F) C νFC)
         (φ : ⟦F⟧ C νFC -> C)
  := fst ∘ out_ ∘ (| out_inv ∘⟨ φ , id ⟩ |).

Notation '' {| φ |} '' := (histo _ _ _ _ _ _ φ).
```

```
Section histomorphism.
  Variable (F : PolyF) (μF : Type) (C : Type) (νFC : Type)
           (ia : F_initial_algebra F C μF)
           (tc : F_terminal_coalgebra (Prod arg1 F) C νFC).

  Proposition histo_charn :
    forall (f : μF -> C) (φ : ⟦F⟧ C νFC -> C),
      f ∘ in_ = φ ∘ F[⟦⟨ f, in_inv ⟩⟧] <-> f = {| φ |}.

  Proposition histo_cancel :
    forall (φ : ⟦F⟧ C νFC -> C),
      {| φ |} ∘ in_ = φ ∘ F[⟦⟨ {| φ |}, in_inv ⟩⟧].
End histomorphism.

Proposition histo_refl (F : PolyF) (μF : Type) (νFC : Type)
            (ia : F_initial_algebra F μF μF)
            (tc : F_terminal_coalgebra (Prod arg1 F) μF νFC)
  : id = {| in_ ∘ F[ fst ∘ out_ ] |}.

(* It is not in principle type contest *)
Proposition histo_fusion :
  forall (F : PolyF) (μF C νFC : Type)
         (ia : F_initial_algebra F C μF)
         (tc : F_terminal_coalgebra (Prod arg1 F) C νFC)
         (φ : ⟦F⟧ C νFC -> C) (ψ : ⟦F⟧ C νFC -> C) (f : C -> C),
    f ∘ φ = ψ ∘ F[ @ana _ C νFC _ νFC ((f ⊗ id) ∘ out_)] -> f ∘ {| φ |} = {| ψ |}.

Proposition histo_cata :
  forall (F : PolyF) (μF C νFC : Type)
         (ia : F_initial_algebra F C μF)
         (tc : F_terminal_coalgebra (Prod arg1 F) C νFC)
         (φ : ⟦F⟧ C C -> C),
    (| φ |) = {| φ ∘ F[fst ∘ out_] |}.

Definition futu (F : PolyF) (νF : Type) (C : Type) (μFC : Type)
         (tc : F_terminal_coalgebra F C νF)
         (ia : F_initial_algebra (Sum arg1 F) C μFC)
         (φ : C -> ⟦F⟧ C μFC)
  := ⟦ [ φ , id ] ∘ in_inv ⟧ ∘ in_ ∘ inl .

Notation ''⟦ φ ⟧ '' := (futu _ _ _ _ _ _ φ).

Section futumorphism.

  Variable (F : PolyF) (νF : Type) (C : Type) (μFC : Type)
```

```
          (tc : F_terminal_coalgebra F C νF)
          (ia : F_initial_algebra (Sum arg1 F) C μFC)
          (φ : C -> ⟦F⟧ C μFC).

  Proposition futu_charn :
    forall (f : C -> νF),
      out_ ∘ f = F[ (| [ f, out_inv ] |) ] ∘ φ <-> f = ⟦⦇ φ ⦈⟧ .

  Proposition futu_charn_onlyif :
    forall (f : C -> νF),
      out_ ∘ f = F[ (| [ f, out_inv ] |) ] ∘ φ -> f = ⟦⦇ φ ⦈⟧ .

  Proposition futu_cancel :
    out_ ∘ ⟦⦇ φ ⦈⟧ = F[ (| [ ⟦⦇ φ ⦈⟧ , out_inv ] |) ] ∘ φ.
End futumorphism.

Proposition futu_refl :
  forall (F : PolyF) (νF : Type) (μFC : Type)
         (tc : F_terminal_coalgebra F νF νF)
         (ia : F_initial_algebra (Sum arg1 F) νF μFC),
    id = ⟦⦇ F[ in_ ∘ inl ] ∘ out_ ⦈⟧.

Proposition futu_fusion :
  forall (F : PolyF) (νF C μFC : Type)
         (tc : F_terminal_coalgebra F C νF)
         (ia : F_initial_algebra (Sum arg1 F) C μFC)
         (φ : C -> ⟦F⟧ C μFC) (ψ : C -> ⟦F⟧ C μFC) (f : C -> C),
    ψ ∘ f = F[ (| in_ ∘ (f ⊗ id) |) ] ∘ φ -> ⟦⦇ ψ ⦈⟧ ∘ f = ⟦⦇ φ ⦈⟧.

Proposition futu_ana :
  forall (F : PolyF) (νF C μFC : Type)
         (tc : F_terminal_coalgebra F C νF)
         (ia : F_initial_algebra (Sum arg1 F) C μFC)
         (φ : C -> ⟦F⟧ C C),
    ⟦⦇ φ ⦈⟧ = ⟦⦇ F[in_ ∘ inl] ∘ φ ⦈⟧.
```